HZ BOOKS 华章IT
[PACKT] PUBLISHING
游戏开发与设计 —技术丛书—

Unity虚拟现实开发实战

[美] 乔纳森·林诺维斯 (Jonathan Linowes) 著 童明 吴迪 译

Unity Virtual Reality Projects

- 本书构建了各种类型的VR体验，其中包括透视图、第一人称角色、过山车、360°投影和社交化的VR。
- 通过项目实践，学习使用Unity开发可以用于Oculus Rift或Google Cardboard等设备体验的VR应用程序。

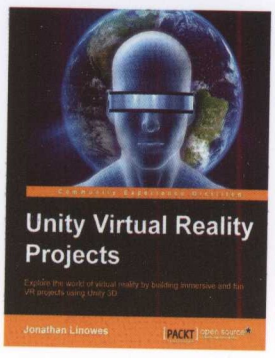
机械工业出版社
China Machine Press

内容简介

本书使用一种实战的、基于项目的方式教你使用Unity开发虚拟现实，不仅提供详细的步骤介绍，还会讨论其中涵盖的更广泛的背景和应用场景。

你将学习使用Unity开发可以用于Oculus Rift或Google Cardboard等设备体验的VR应用程序。然后，学习从第三人称和第一人称的视角融入虚拟世界。另外，你将探索一些可能是VR才有的且十分重要的技术。本书中的项目将演示如何构建各种VR体验，你可以通过交互式的Unity编辑器和C#编程深入Unity 3D引擎。阅读完本书后，你将有能力使用Unity开发丰富的、交互式的虚拟现实体验程序。

原书封面



游戏开发与设计
——技术丛书——

Unity虚拟现实开发实战

[美] 乔纳森·林诺维斯 (Jonathan Linowes) 著 童明 吴迪 译

Unity Virtual Reality Projects

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Unity 虚拟现实开发实战 /（美）乔纳森·林诺维斯（Jonathan Linowes）著；童明，吴迪译 . —北京：机械工业出版社，2016.10
（游戏开发与设计技术丛书）
书名原文：Unity Virtual Reality Projects

ISBN 978-7-111-55131-7

I. U… II. ① 乔… ② 童… ③ 吴… III. 游戏程序—程序设计 IV. TP317.6

中国版本图书馆 CIP 数据核字（2016）第 249196 号

本书版权登记号：图字：01-2016-3486

Unity 虚拟现实开发实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：李　静
责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司
版　　次：2016 年 11 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16
印　　张：14.25
书　　号：ISBN 978-7-111-55131-7
定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379246　88361066
购书热线：（010）68326294　88379649　68995259
投稿热线：（010）88379604
读者信箱：hzit@hzbook.com

The Translator's Words 译者序

毫无疑问，虚拟现实（VR）是科技界2016年最热的关键词。

这不仅是因为虚拟现实技术能够改变我们娱乐和游戏的方式，而且对于很多行业或领域都具有重要意义。比如，旅游业、电影行业、视频会议行业、医学行业、室内设计、房地产开发、军事演练、工业仿真、应急演练、文物古迹展示、教育、康复训练等都将可以应用虚拟现实技术增强用户体验，为业界带来巨大的价值。

而虚拟现实本身也包括一条很长的产业链，从虚拟现实硬件设备的制造到软件系统的研发，从操控技术到核心算法，从虚拟现实内容的采集到虚拟现实内容的生产，这些都带来了大量新的创业和就业机会。

得益于移动互联网和智能硬件的浪潮，软硬件技术成熟，消费级虚拟现实所需硬件设备的价格能够被普通人接受，最便宜的Google Cardboard仅用几十块钱就可以拥有。未来几年，虚拟现实必将快速融入我们的生活。

而对于对虚拟现实感兴趣的我们，最令人兴奋的事当然是通过学习参与创建自己的虚拟现实应用了。虚拟现实的关键技术点包括3D图形渲染、第一人称操控、头部运动跟踪、用户输入控制、镜头变形、全景场景、晕动症处理等。Unity 3D界面友好、上手较快，是当今最流行的3D引擎之一，也是最早支持虚拟现实的引擎之一，而Cardboard是成本最低廉的虚拟现实设备。当然，学习这些技术点最快速的开始方式是动手使用Unity 3D结合Google的Cardboard和手机开发简单的虚拟现实应用。

本书以循序渐进、深入浅出、具体案例讲述的方式引导读者走进虚拟现实的世界，相信读完此书后，读者不仅能够理解虚拟现实的基本原理并了解虚拟现实的关键技术点，还能够从书中的范例中学习和总结，进而结合Unity 3D引擎创建出自己卓越的虚拟现实应用。

即使你不是开发人员，也不熟悉 3D 图形学，仅对虚拟现实兴趣浓厚，原作者书中对于虚拟现实的深入理解也能够使你受益匪浅。

有幸翻译本书，非常感谢缪杰编辑和李静编辑对我的信任和帮助，感谢我的翻译伙伴吴迪的共同努力，感谢我的妻子对我的支持。

童明

2016 年 8 月于北京

About the Reviewers 审校者简介

Krystian Babilinski，中学时期就开始使用 Unity 和 Blender 工作，从高中起，他和他的弟弟 Adrian 开始在 Google 的 Helpouts 服务上学习。在学习的过程中，将自己暴露于新的问题集中，并亲自解决这些问题。慢慢地，他开始接触大规模的项目并成为自由职业者。凭借着不断成长以及对 Unity 3D 平台优化的知识，他和弟弟在 2014 年创办了自己的创新公司，他们开始为 Unity 的资源商店开发资源，后来为更大的客户服务，像 Hasley Group 和 Beach Consulting，没有这些忠实的客户他们不会有如今的这些成功。

Arindam Ashim Bose，自 2015 年起，就一直奋斗在获取亚特兰大乔治亚理工学院计算机科学硕士学位的路上。他对计算机图形、虚拟现实和增强现实以及游戏开发很有兴趣。

他出生在孟买，从少年起就着迷于计算机技术，尤其是计算机游戏。他在假期中花费无数的时间玩游戏并且摆弄这些游戏以便修改它们。摆弄和修改的兴趣让他深入到计算机编程之中。

他目前正在努力进入游戏行业并准备成为一名程序员，同时也在努力获取硕士学位。

Rongkai Guo，肯纳索州立大学计算机软件工程和游戏设计开发学院的一名助教。他的研究兴趣是严肃游戏（serious gaming）、计算机或手机游戏和虚拟现实（VR）。他四年多来一直在指导研究一个用于康复的 VR 项目。他还负责了他的第一个基础研究——IN THE WORLD，此项目正式地研究了 VR 如何影响行动不便的人。

Arun Kulshreshth，中佛罗里达大学计算机科学学院的一名研究员。他的研究兴趣是 3D 用户界面、人机交互（HCI）、游戏和虚拟现实。他于 2005 年在德里获取了印度理工学院的数学与计算科学的工程硕士学位。他于 2012 年获取了中佛罗里达大学的计算机

科学硕士学位，并在 2015 年获取了该大学的博士学位。

他是几个视频游戏领域（比如立体 3D、头部跟踪、手势接口等）相关 3D 用户界面技术出版物的作者。他还是一位国际计算机会学（Association of Computing Machinery，ACM）和电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）的专家。在过去的时间里，他在世界各地指导研究，包括西班牙、丹麦和美国。他其中的一篇论文在 HCI 大会（CHI 2014）上获得了优秀奖。2014 年，他的名字出现在路透社的文章中，并且他的一个项目被 Discovery News 报道。

Robin de Lange，一名研究人员、讲师及专注于虚拟现实和教育的企业家。Robin 拥有莱顿大学多媒体技术的硕士学位以及物理学与哲学的学士学位。他在导师 Bas Haring 带领下的多媒体技术研究组完成了博士科研。他的科研内容是探索增强现实和虚拟现实的潜能以便理解和解决复杂的问题。这个研究中的一部分是一门选修课，在该课程中 Robin 领导一组学生完成了用于教育领域的虚拟现实的原型。

除了他的学术生涯以外，Robine 还参与了很多不同的活动。他是一个教育辅导机构的总监、Lyceo CodeWeken 的创始人，还是一位教授高中学生如何写代码的独立策划人。

Samuel Mosley，一位游戏设计师，他在达拉斯的得克萨斯大学研究编程和游戏设计。他的兴趣在编程和游戏上，他希望在这两个领域都能够扮演重要的角色。他目前作为一名游戏设计师就职于 Bohemia Interactive Simulations。

Preface 前　言

如今，我们正见证着虚拟现实（VR）的迅猛发展，这是一项令人激动的技术，它有望改变我们与信息、朋友和整个世界进行交互的基本方式。

什么是消费级虚拟现实？戴上一个头盔显示器（比如护目镜），你可以观看立体 3D 场景，你可以通过移动头部向四周看以及通过使用手持控制或动作传感器四处走动，你可以拥有完整的沉浸式体验。这就像你真的在某个虚拟世界中一样。

本书通过实战的、基于项目的方式教你使用 Unity 3D 游戏引擎开发虚拟现实的细节。我们通过一系列实战工程、循序渐进的教程，并使用 Unity 5 和其他免费或开源软件进行深入讨论。而 VR 技术正在快速发展，我们将试着获取基本的原则和技巧，以便使你能够让自己的 VR 游戏和应用程序具有沉浸感和舒适感。

你将学习如何使用 Unity 开发可以用 Oculus Rift 或者 Google Cardboard 这样的设备体验的 VR 应用程序。我们将涵盖技术上对于 VR 来说重要且可能独一无二的考虑。读完本书后，你将有能力用 Unity 开发丰富的、交互性的虚拟现实体验程序。

本书的主要内容

第 1 章介绍消费级虚拟现实（VR）中关于游戏和非游戏应用程序的新技术和新机遇。

第 2 章讨论如何构建一个简单的透视图场景。本章介绍了 Unity 3D 游戏引擎和用于 3D 建模的 Blender，并探索了世界坐标系和缩放比例的问题。

第 3 章帮助你配置项目以便运行于 VR 头盔，比如 Oculus Rift 和 Google Cardboard（Android 或 iOS）。然后，我们深入了解 VR 硬件和软件如何运行的细节。

第 4 章探讨了场景中的 VR 摄像机与对象的关系，包括 3D 光标与基于凝视的射线

枪。本章还包括使用 C# 编程语言编写 Unity 脚本。

第 5 章实现了很多 VR 的用户界面（UI）例子，包括一个平视显示器（HUD）、信息框，以及带有很多代码和说明的游戏中的对象。

第 6 章剖析 Unity 的人物角色对象和组件，它们用于构建我们自己的带有基于凝视的导航功能的第一人称角色。然后，我们将探讨拥有一个第一人称虚拟身体的体验，并考虑晕动症的问题。

第 7 章在我们通过若干个 VR 项目案例和游戏学习使用作用力和重力的同时，深入了解 Unity 的物理引擎、组件及材质。

第 8 章帮助我们构建一个 3D 架构的空间和实现虚拟的漫游。我们还讨论了 Unity 中的渲染和性能优化。

第 9 章利用全部的 360° 在各种项目中使用 360° 多媒体，其中包括地球仪、全景图、照片球。我们还讨论了它们的原理。

第 10 章探讨了使用 Unity 5 的网络组件带 VR 功能的多玩家实现。我们还看了 VRChat 作为一个用于社交式 VR 的可扩展平台的例子。

第 11 章展望 VR 技术。

准备工作

在我们开始前，你将需要完成一些工作。随便吃点东西，一瓶水或一杯咖啡。除了这些，你还需要一台安装了 Unity 3D 游戏引擎的 PC（Windows 或 Mac）。

你不需要超级性能的配置，因为 Unity 可以像猛兽一样渲染复杂的场景，而 Oculus 已经发布了推荐的 PC 硬件规格，你可以用较少的钱获取到。甚至一台笔记本就可以完成本书中的项目。要想获取 Unity，访问 https://unity3d.com/get-unity/，选择你想用的版本，点击 Download Installer，然后继续按照说明操作即可。选择 Unity 的免费个人版本就行。

我们还可以选择使用 Blender 开源项目进行 3D 建模。本书不是关于 Blender 的著作，但是如果你需要的话，我们可以使用它。要获取 Blender 可以访问 http://www.blender.org/download/，根据你的系统平台按照说明下载。

建议拥有一台虚拟现实头盔显示器（HMD）以便试运行你的构建成果以及获取本书中项目开发的第一手体验。完全可以在一台台式机显示器上构建和运行所有的项目，但是乐趣在哪里呢？本书将涉及 Google Cardboard 和 Oculus Rift 的细节。

Google Cardboard 是移动 VR 的一个示范，让你可以用智能手机运行 VR 应用。如果你有一台 Android 智能手机，你需要从 Google 获取 Android 开发工具；如果你有一台 iOS 设备，你需要从 Apple 获取 Xcode 开发工具（及授权）。具体细节在第 3 章有所介绍。

Oculus Rift 是一个关于 Desktop VR 的例子目前，Unity 对 Rift 内置支持。然而，如果你有不同的头盔显示器，你可能还需要从厂商处下载一个 Unity 的接口包。具体细节也在第 3 章中进行介绍。

这应该就差不多了——一台 PC、Unity 软件、一台 HMD，我们就准备好了！

目标读者

如果你对虚拟现实感兴趣，想要学习它的原理或者想创建自己的 VR 体验，那么本书适合你。不管你是否是程序员，是否熟悉 3D 计算机图形，或者有前两者的经验但对于 VR 是个新手，你都可以从本书中受益。使用 Unity 并不是重新开始，但你也不需要是一位专家。然而，如果你是 Unity 新手，只要你认为自己可以适应本书的节奏，就可以购买本书。

游戏开发者可能已经熟悉本书中的概念了，在学习了很多特定于 VR 的知识后把它们重新应用于 VR 项目。已经了解如何使用 Unity 的移动平台和 2D 游戏的设计师将会发现另一个维度！工程师和 3D 设计师可能理解很多 3D 概念，但是他们可以学习把游戏引擎用于 VR。应用程序开发者也许会感激 VR 在非游戏用途方面的潜能，并且想要学习使用制作这类程序的工具。

约定

本书中，你将发现很多文本风格以区分不同种类的信息。这里有这些风格的一些例子和它们含义的说明。

代码会设置成下面这样：

```
using UnityEngine;
using System.Collections;

public class RandomPosition : MonoBehaviour {
  // Use this for initialization
  void Start () {
  }
```

```
    // Update is called once per frame
    void Update () {
    }
}
```

当我们想让你注意代码块的某一部分时，相关的行或项会加粗：

```
public class ButtonExecute : MonoBehaviour {
    public float timeToSelect = 2.0f;
    private float countDown;
    private GameObject currentButton;
    private clicker = new Clicker ();
```

任何命令行输入或输出都写成下面这样：

```
moveDirection *= moveDirection * velocity * Time.deltaTime;
transform.position += transform.position  + moveDirection;
```

新的术语或**重要的词语**以黑体显示。你在屏幕上看见的单词，举个例子，在菜单或对话框中，文本中这样显示：

“点击这里**创建一个 Room 按钮**。”

表示警告或重要的注意事项。

表示技巧或小巧门。

读者反馈

来自我们读者的反馈永远是受欢迎的。让我们知道你对于本书的看法——你喜欢与否。读者反馈对我们来说很重要，因为它帮助我们改进那些你们可以真正利用的内容。

若想向我们发送常规的反馈，请写电子邮件到 feedback@packtpub.com，并在邮件主题中提及书名即可。

如果你对某个话题有专业意见或者你对写作或参与一本书的编写有兴趣，请参见我们的作者指南：www.packtpub.com/authors。

客户支持

你已经是 Packt 图书的尊贵用户，我们有很多方式帮助你从购买中获取最大价值。

下载代码示例

你可以通过在 http://www.packtpub.com 的账户下载所有你在 Packt 出版社所购买书籍的示例代码文件。如果你是在其他地方购买的本书，可以访问 http://www.packtpub.com/support 并注册，以便我们通过电子邮件把文件发送给你。

下载本书的彩色插图

我们还向你提供本书中所用到的截图 / 图表的彩色插图的 PDF 文件。这些彩色插图将帮助你更好地理解输出结果中的改变。你可以从 http://www.packtpub.com/sites/default/files/downloads/ 8556OS_ColorImages.pdf 下载这个文件。

勘误

尽管我们已经尽力去确保内容的准确性，但是错误仍在所难免。如果你发现了书中的错误，也许是文本或代码，若能报告给我们，我们将非常感激。因为这么做，你可以帮助其他读者，并且能够帮助我们改进本书接下来的版本。如果你发现了任何错误，请通过 http://www.packtpub.com/submit-errata 报告它们，选择书名，点击“勘误提交表（Errata Submission Form）”链接，然后键入你的勘误细节。一旦你的勘误被确认，你的提交将会被采纳，而这个勘误将会被上传到我们的网站或添加到之前标题下的勘误部分的任何现在已有勘误表中。

要审阅之前提交的勘误，请访问 https://www.packtpub.com/books/content/support，并在搜索框中输入书名。所需要的信息就会出现在勘误部分之下。

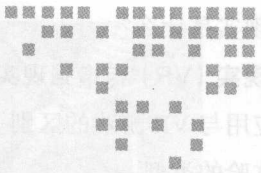
盗版行为

盗版行为在互联网上是一个持续存在的问题，对于所有媒体皆如此。在 Packt，我们非常重视保护我们的版权和许可。如果你在互联网上遇到以任何形式非法复制我们的产品，请立即向我们提供地址或网站名称，以便我们能够弥补这一缺失。

请通过 copyright@packtpub.com 联系我们，并附上有盗版嫌疑的材料链接。

我们非常感激你在保护作者以及我们向你提供有价值的内容上所做的帮助。

目　录 *Contents*



第 1 章 Chapter 1

万物皆可虚拟

“虚拟现实”这一概念让人们对于“在某个地方”所表达的意思产生了疑问。

在手机发明之前，如果你打电话给某个人，“嘿，你在哪呢？”这样说没有任何意义。因为你知道他们在哪里，你打给他们家，他们就在家里。

然而在手机开始普及后，你开始听到人们开始这么说：“你好。哦，我现在在星巴克。”手机另一端的人并不能非常确切地知道你在哪里，因为你的声音不再跟你的房子联系在一起了。

说起 VR，我有这样一个例子：当我回到家，我妻子把孩子们安顿下来之后，她便有一些属于自己的时间，于是她坐到沙发上，带上护目镜。这时我走过来，拍拍她的肩膀，这样问道：“嘿，你现在在哪里？”

这太奇怪了。一个人就坐在你的面前，你却不知道他在哪里。

——Jonathan Stark，移动专家 & 播主

欢迎来到虚拟现实的世界！本书将探讨如何创建属于自己的虚拟现实体验。我们将会涉及一系列实践项目、循序渐进的教程，并深入探讨如何使用 **Unity 5** 3D 游戏引擎和其他免费或者开源软件。虽然虚拟现实技术发展迅速，但我们尽量尝试只掌握那些可以让 VR 游戏和应用更有沉浸感和舒适感的基础原理和技术。

本章将定义虚拟现实，并且举例说明它是如何应用在游戏和其他有趣的场景和产品上的。本章将讨论如下话题：

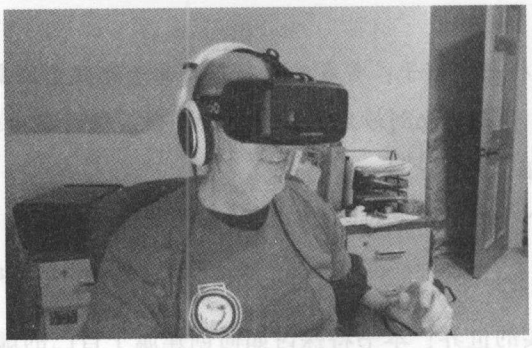
- 虚拟现实是什么？
- **虚拟现实（VR）与增强现实（AR）的区别。**
- VR 应用与 VR 游戏的区别。
- VR 体验的类型。
- 开发 VR 必备的技能。

1.1 虚拟现实对你来说意味着什么

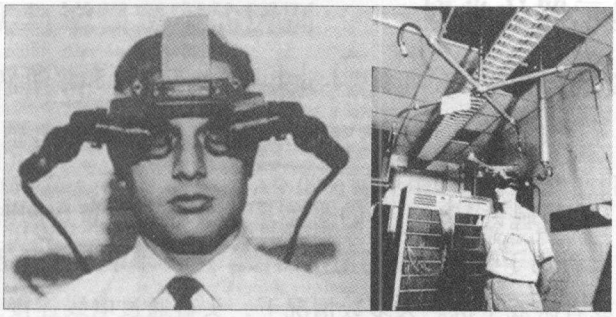
如今，我们正见证着消费级虚拟现实的迅猛发展，这是一项令人激动的技术，它有望改变我们与信息、朋友和整个世界进行交互的基本方式。

什么是虚拟现实？通常，VR 是使用特殊的电子设备，由计算机生成的对 3D 环境的模拟，对于正在体验它的人来说，它看起来非常真实，目标是达到一种处于虚拟环境中的强烈感觉。

现在的消费级 VR 通过戴上头盔显示器（比如护目镜）观察立体的 3D 场景。你可以通过移动头部观察四周，并且通过手持控制器或者动作传感器向周围走动。你会被带入沉浸感十足的体验当中，就像真正处在某个虚拟世界中一样。下图展示了一个人正在体验 **Oculus Rift Development Kit 2（DK2）**：



虚拟现实并不是一项新事物。虽然它被隐藏在某些学术研究实验室和高端产业及军事设施中，但早在几十年前就已经存在。过去它非常庞大、笨重，并且昂贵。Ivan Sutherland 在 1966 年发明了第一台头盔显示器，如下图所示。它被吊在天花板上！在过去的一段时间内，一些将消费级虚拟现实产品上市的尝试都失败了。



“终极显示”，Ivan Sutherland, 1965

在 2012 年，Palmer Luckey，Oculus VR 有限责任公司的创始人，将一个尚未开发完毕的 VR 头盔显示器交到了 John Carmack——Doom、Wolfenstein 3D 和 Quake 经典游戏的著名开发者的手里。他们一同在 **Kickstarter**（著名众筹网站）上发起众筹，并且在一个狂热的社区中发布了一个开发套件，该套件称为 Oculus Rift Development Kit 1(DK1)。这引起了包括 Mark Zuckerberg 在内等投资人的注意，并于 2014 年，Facebook 以 2 亿美元的价格收购了这个公司。没有产品，没有用户，只有永恒的承诺，它所吸引的资金和引起的关注已经帮助一个新型的消费级产品火热起来。其他公司也开始跟进，其中包括 Google、Sony、Samsung 和 Steam。增强 VR 体验的创新和设备层出不穷。

大多数的基础研究已经完成，技术也已经成熟，这很大程度上要归功于运行移动技术设备的大规模普及。有一个庞大的开发者社区，他们对于构建 3D 游戏和手机应用非常有经验。创意内容生产商也加入进来，媒体对它的讨论也越来越多。终于，虚拟现实成为现实。

说什么？虚拟现实成为了现实？哈！如果它是虚拟的，它怎么能……哦，别在意。

最后，我们会将注意力从新兴的硬件设备上转移，并且认识到“内容为王”。现在的 3D 开发软件（商业的、免费的和开源的）催生了大量的独立开发团队或个人，游戏开发者同样可以用它们来创建非游戏 VR 应用。

尽管在游戏领域 VR 获得了很多的狂热追求者，但是在更有潜力的应用领域，它将拥有更多的拥趸。对于目前正在使用 3D 建模和计算机制图的任何业务，如果开发者使用 VR 技术将会变得更高效。由 VR 所赋予的沉浸感能够增强所有常见的线上体验，其中包括：工程领域、社交网络、购物、营销、娱乐和业务拓展。在不久的将来，带着 VR 头盔设备浏览 3D 网站可能与现在访问平面网站一样普遍。

1.2 头戴式显示器的类型

目前，虚拟现实头盔显示器有两个基本分类——**桌面 VR 和移动 VR**。

1.2.1 桌面 VR

对于桌面 VR（或控制台 VR），你的头盔是一个外围设备，同时它还有一台更强大的计算机作为主设备用于处理大量的图形图像。这台计算机可以是一台 Windows 主机、MAC、Linux 或者游戏主机。大多数情况下，头盔通过电线连接到计算机，游戏运行在远程机器上，而**头盔显示器**（Head-mounted Display，HMD）是具有动作感应输入功能的外围显示设备。术语“桌面”有些用词不当，因为它很有可能被放在客厅或者书房中。

Oculus Rift（Https://www.oculus.com/）是一个在眼镜上集成了显示器和传感器设备的范例。游戏运行在一台单独的主机上。其他的桌面头盔设备还有 **HTC** 的 **Valve Vive** 和索尼用于 PlayStation 的 **Morpheus** 项目。

Oculus Rift 与台式计算机通过视频线和 USB 线进行连接，通常，**图像处理单元**（Graphics Processing Unit，GPU）越强大效果越好。然而，对于本书来说，在我们项目中并没有太多繁重的渲染工作，你用笔记本电脑就可以（需要有两个 USB 端口和一个 HDMI 端口可用）。

1.2.2 移动 VR

以 **Google Cardboard**（http://www.google.com/get/cardboard/）为例，移动 VR 是一台包含了两个透镜和一个手机插槽的简单装置。手机的显示屏用于显示两个立体视图。它可以追踪头部旋转，但是并不能捕捉位置。Cardboard 同样为用户提供了点击或者轻按边缘以实现在游戏中进行选择的功能。它对要处理图像的复杂度有一定的限制，因为它是通过使用你的手机处理器来把视图渲染在手机显示屏上。其他的移动 VR 头盔设备包括 Samsung 的 **Gear VR** 和 Zeiss 的 **VR One** 等。

Google 提供了开源规范，而有些生产厂商已经开发了一些现成的模型进行售卖，价格同样低至 15 美元。如果你想找一种，Google 一下吧！现在有多个版本的 Cardboard，兼容了各种型号的手机，包括 Android 和 iOS。

尽管使用 Cardboard 设备进行 VR 体验的品质有限（甚至有人说是不能胜任的），它可能是一个“入门级”设备，在若干年后将会成为古董，然而 Cardboard 还是适用于本书的一些小型项目，后面我们会不时地来回顾它的局限。

1.3　虚拟现实与增强现实的区别

也许该花点时间澄清一下虚拟现实不是什么了。

VR 有一项兄弟技术是增强现实（AR），它在现实世界的视图上叠加了**计算机成像**（Computer Generated Imagery，CGI）。AR 的有限使用可以在智能手机、平板、手持游戏系统（例如任天堂 3DS），甚至在一些科学博物馆的展览中被发现。它将 CGI 叠加在来自摄像机的实时视频上。

最新的 AR 创新是 AR 头盔，例如微软的 **HoloLens** 和 **Magic Leap**，它们直接将计算机图形在你的视野中显示出来，而并不是融合进视频图像中。如果把 VR 头盔比作密封的护目镜的话，那 AR 头盔很像半透明的太阳镜，它使用了称为**光场**的技术，可以将真实世界的光线与计算机成像进行融合。对于 AR 来说，有一项挑战是如何保证计算机成像要时刻与现实世界中的物体保持一致且完全映射在其表面，同时在走来走去的时候消除延迟感并让它们（计算机成像与现实世界中的物体）仍然保持一致。

在未来应用的场景中，人们对 AR 抱有同 VR 一样的期待，但是它们还是有所不同。这是因为 AR 致力于使用户融入到他们当前的环境中，而 VR 是完全沉浸式的。在 AR 体验场景中，你可以打开双手并且能够看到一个小木屋正位于你的手掌上；但是在 VR 中，你可能会被传送到木屋内部并且你可以在它里面随意走动。

我们同时非常期待融合 AR 和 VR 的混合设备，或者可以在这两种模式间切换的设备。

1.4　应用与游戏

消费级的虚拟现实从游戏开始。玩家早就已经习惯于具有高度交互的超现实的 3D 场景了。VR 仅仅是更进了一步。

玩家是高端图像技术的早期体验者。数以百万计的游戏机和基于 PC 的组件大规模生产以及供应商之间的竞争带来了更低的价格和更高的性能。同样，游戏开发者也经常推动技术的发展，极力释放硬件和软件的性能。玩家是一个非常苛刻的群体，而市场又紧跟他们的需求。这一波 VR 硬件和软件公司中的大多数，也许不是全部，大都会将视频游戏行业作为首要目标，这一点不足为奇。在 **Oculus Share**（https://share.oculus.com/）和 Cardboard 应用的 **Google Play**（https://play.google.com/store/search?q=cardboard&c=apps）上，示例和可下载的主要都是游戏。游戏玩家是最热情的

VR 拥护者，并且非常欣赏它的潜力。

游戏开发者们明白，一个游戏的核心是**游戏玩法**，或者说游戏规则，它很大程度上与外观或者说是游戏的主题无关。游戏玩法可以包括解谜、运气、策略、计时或者肌肉记忆 。VR 游戏能够拥有同样的玩法元素但是需要对虚拟场景做出一些调整。举例来说，第一人称角色在主机视频游戏中行走，他的步频大概是他在真实世界中的 1.5 倍。如果不这样做，那么玩家将会感觉到这个游戏太慢并且有些无聊。将同样的角色放到 VR 场景中他们反而会觉得太快，这将导致玩家的反感。在 VR 游戏中，你会想让你的角色以正常速度行走。并不是所有的视频游戏都能很好地对应到 VR，当你真正处在一个战区中间的时候，就不那么有趣了。

也就是说，虚拟现实同样可以应用在除游戏之外的区域。尽管游戏还将同样重要，但是非游戏应用最终将超过它们。这些应用在很多方面与游戏不同，最具标志性的是，应用不太注重游戏玩法反而更在意自身体验和应用的特定目标。当然也不排除有一些游戏玩法。举例来说，某个应用专门用于培训用户的某项特殊技能。有时，商业或者个人应用的游戏化会使它更有趣，并且更有效地通过竞争驱动所期望的行为。

通常，非游戏 VR 应用不太在意输赢而更多地注重本身的体验。

下面是现在人们正在做的非游戏应用的几个类型：

- **旅游与观光**：足不出户即可访问遥远的地方。用一下午的时间访问巴黎、纽约和东京的艺术博物馆。在火星散步。甚至你可以坐在你在佛蒙特州（Vermont）寒冷的小屋里享受印度的胡里节（丰富多彩的春节）。
- **机械工程和工业设计**：计算机辅助软件，例如在三维建模、模拟和可视化方面的先驱 AutoCAD 和 SOLIDWORKS。配合 VR，工程师和设计师们将可以在手持终端设备真正修建之前直接亲身体验并且以低成本在假设场景使用。考虑一个新汽车设计的迭代。它的外观看起来怎样？它的性能如何？在驾驶位上会是怎么样的体验？
- **建筑与土木工程**：仅仅是为了向客户和投资者推销他们的想法，或者更重要的是，去验证许多设计的假设，建筑师和工程师们经常会构建他们设计的一个缩小模型。现在，建模和渲染软件经常用于从建筑平面图构建虚拟模型。通过 VR，与利益相关方的谈判将更有信心。其他人员，例如室内设计师、暖通空调设计师（HVAC）和电力工程师，也会很快加入进来。

- **房地产**：房地产经纪人一直是互联网和可视化技术的快速采用者，用以吸引买家并完成销售。房地产搜索网站是网络的第一次成功使用。待售物业的在线全景巡查视频现在已经非常普遍。配合 VR，你人在纽约就可以在洛杉矶找到住处。如果使用移动 3D 感应技术，例如谷歌的 **Project Tango**（https://www.google.com/atap/projecttango），这将会变得更容易，它提供了使用手机对房间进行一次 3D 扫描而后自动对空间构建出一个模型的功能。
- **医学**：VR 对于健康和医疗的潜力简直有生与死那么重要。每天，医院使用 MRI 和其他的扫描设备为我们的骨骼和器官生产模型，用于医疗诊断或者术前计划。使用 VR 来提高可视化和测量将会提供更直观的分析。虚拟现实还可以用于手术模拟来训练医学专业的学生。
- **心理健康**：已经证明，虚拟现实体验已经在**暴露疗法**（exposure therapy）中对于治疗**创伤后应激障碍**（Post Traumatic Stress Disorder，PTSD）是有效的，病人在心理治疗师的指引下，通过对经历的复述来面对他们创伤的回忆。类似地，VR 还被用于治疗蜘蛛恐惧症和对飞行的恐惧。
- **教育**：虚拟现实在教育领域的机会实在是太明显了。第一个成功的 VR 体验是 **Titans of Space**，它让你掌握探索银河系的第一手资料。自然、历史、艺术和数学——VR 将帮助全年龄段的学生，就像他们所说的，读万卷书不如行万里路。
- **培训**：丰田已经展示了一个针对驾驶员教育的 VR 模拟，用于教育青少年有关不集中注意力驾驶的危害。在另一个项目中，职业学校的学生们可以体验操作起重机和其他重型建筑设备。训练急救人员、警察、消防和救援人员的培训可以通过虚拟现实呈现高风险情况和可替代的虚拟场景来提升技能。NFL 正寻求通过 VR 进行的运动员训练。
- **娱乐和新闻**：虚拟地参与摇滚现场和体育赛事，观看音乐视频。就像你在现场一样重新体验新闻事件。享受 360° 的电影体验。讲故事的艺术将会被虚拟现实改变。

哇，这真是个长长的列表啊！而这些仅仅是很容易实现的东西。

本书的目的并不是要过于深入地讨论这些应用。相反，我希望这个纵览有助于激发你的思考，并且提供一个关于如何将万物虚化的视角。

1.5 本书涵盖的内容

本书采用一个实践性的、基于项目的方法教授使用Unity3D游戏开发引擎开发的虚拟现实细节。你将学习如何使用Unity 5开发可以使用诸如Oculus Rift 和Google Cardboard等设备体验的VR应用。

然而，我们还有一个小问题——这项技术发展得太迅速了。当然，这是个甜蜜的负担。实际上，这是个非常棒的问题，除非你是一个项目开发者或者一个关于这项技术的书籍作者！如何做到写一本书，保证在它出版的时候没有过时的内容呢？

本书中，我试图提炼出一些基本原理，它们会比任何近期的虚拟现实技术的发展存在得更久，其中包括以下内容：

- 用示例项目对不同的VR体验进行分类。
- 重要的技术思路和技能，特别是与构建VR应用相关的。
- VR设备和软件工作原理的一般性解释。
- 保证用户舒适度和降低VR晕动症的策略。
- 介绍使用Unity游戏引擎创建VR体验。

一旦VR变得主流，大多数章节将有可能变得显而易见而非过时，就像在今天看来，我们会觉得20世纪80年代的人们讨论怎么使用鼠标一样奇怪。

你是谁

如果你对虚拟现实感兴趣，想要学习它是如何工作的，或者想自己创建VR体验，那么本书就是为你而写。我们将指导你完成一系列手把手的项目，循序渐进地讲授教程，并深入地讨论如何使用Unity 3D游戏引擎。

无论你是一名熟悉3D计算机图形学的非编程人员，或者你对编程和图形学非常有经验但是对于虚拟现实很陌生，你同样将从本书中获益匪浅。你需要了解一些Unity知识，但不必要是专家。同样，如果你不熟悉Unity，只要你觉得自己能够适应本书的节奏，就可以将它拿来参考。

游戏开发者可能已经熟悉本书中的一些概念，如果学习了一些特定于VR的思路，这些概念将同样适用于VR项目。工程师和3D设计师可能了解许多3D概念，但是他们可能希望学习如何使用游戏引擎来体验VR。引用开发者们可能意识到VR在非游戏领域的潜力并且想要学习一些让这些实现的工具。

无论你是什么角色，我们都打算将你变成一个3D软件VR忍者。好了，这是本书的

延伸目标，但我们会带你上路。

1.6　VR 体验类型

虚拟现实体验的种类不止一种。事实上，有很多种，可以考虑虚拟现实的下述类型：

- **透视图**：最简单的情况是，我们自己创建一个 3D 场景。你正在通过第三人称视角对其进行观察。你的眼睛就是一部摄像机。实际上，你的每一只眼睛都是一部单独的摄像机，并且能够提供给你一个立体视角。你可以向四周看。
- **第一人称体验**：这种情况下，你沉浸在场景中，并且作为一个可以自由移动的角色。通过使用输入控制器（键盘、游戏手柄或其他技术），你可以四处走动并探索这个虚拟场景。
- **交互式虚拟环境**：这个有点类似于第一人称体验，但是它还有一个附加的特性——当你处在这种场景中时，你可以与其中的物体交互。这是物理学在起作用。这些物体有可能会响应你。你可能会需要达到某些特别的目标或者完成一些具有游戏规则的挑战。你可能会获得积分或者保持比分。
- **过山车式**：在这种体验中，你是坐着的并且在环境中穿梭（或者是，环境在你周围变化）。举个例子，你可以通过这种虚拟现实体验过山车。但是，它不一定是一次惊悚之旅。它也有可能是一个简单的房地产虚拟漫步或者甚至是慢速的、简单的、冥想的体验。
- **360° 媒体**：想象一下通过 GoPro 拍摄的全景图片被投影在一个球体内部。你处在球体中并可以向四周看。一些纯粹主义者并不认为这是“真实的”虚拟现实，因为你正在看着一个投影而不是一个模型渲染。然而，它可以提供更有效的存在感。
- **社交 VR**：当多个玩家角色进入同一个 VR 场景中并且可以相互看到以及相互对话的时候，它将变成一次令人印象深刻的社交体验。

本书中，我们将会实现几个项目用于展示如何构建这些类型的 VR 体验。为了简便起见，我们需要保持它的纯粹和简单，建议做进一步的调查。

1.7　VR 必备技能

本书每章都介绍了一些对于构建属于自己的虚拟现实应用非常重要的新技巧和概念。

你可以在本书中学习到以下内容：

- **空间缩放比例**：当构建一个 VR 体验时，重视 3D 空间和缩放是很重要的。Unity 的 1 个单位通常等于虚拟世界中的 1m。
- **第一人称控制**：有很多种技术可以用来控制你的虚拟角色（第一人称摄像机）的移动，基于凝视的选择器、游戏控制器和头部移动。
- **UI 控件**：不同于传统的视频游戏（和手机游戏），所有的 UI 组件都处于 VR 中的世界坐标系中，而不是屏幕坐标系。我们将探讨如何向用户显示提醒、按钮、选择器和其他 UI 控件，这样他们就可以进行交互并做出选择。
- **物理和重力**：虚拟现实中的存在感和沉浸感的关键是现实世界中的物理和重力。我们将利用 Unity 的物理引擎。
- **动画**：场景中移动的物体称为“动画”，它可以沿着预定的路径，也可以使用人工智能（Artificial Intelligence，AI）脚本，脚本遵循某种逻辑算法用于响应环境中的事件。
- **多用户服务**：实时联网和多用户游戏不容易实现，但是在线服务使之变得容易，并且不需要你是一个计算机工程师。
- **构建并运行**：不同的头盔显示器使用不同的开发套件（SDK）和资源来构建针对不同意图的应用。我们将考虑对于不同的设备使用一个接口的技术。

在需要的时候我们会使用 C# 语言编写脚本以及使用 Unity 的特性。

然而，有些技术领域我们没有涉及，例如真实感渲染、着色器、材质和光照。我们将不会探究建模技术、地形和骨骼动画。有效地使用高级的输入设备和手部、身体追踪对 VR 来说是至关重要的，但是我们此处不讨论它们。我们同样也不讨论游戏玩法、动力学和策略。我们会讨论渲染性能优化，但不会太深入。这些都是非常重要的话题，可能对你来说是必须要学习的（或者对于你团队中的某些人），此处，它们对于构建完整的、成功的和沉浸感极强的 VR 应用也很重要。

小结

本章中，我们简单地了解了虚拟现实，并且意识到对于不同的人来说它意味着很多事情，并且可以开发出不同的应用。它没有唯一的定义，它在持续变化。但是我们并不孤单，因为很多人都在试图弄明白它。事实上，虚拟现实成为一种新型媒介将在数年后，

但或许又用不了十年，它将发挥自身的潜力。

VR 并不是仅仅用于游戏，它同样可以改变各种不同的应用。我们已经构建了很多个应用。有很多种不同的 VR 体验，我们将在本书中通过工程进行探索。

VR 头盔设备可以分为两类：一种需要单独的处理器（例如一台台式 PC 或主机），它运行一个强大的 GPU；另一种使用你的手机进行处理。本书中，我们将使用 Oculus Rift 作为台式 VR 的例子，而使用 Google Cardboard 作为移动 VR 的例子，虽然现在还有很多其他可替代的新设备。

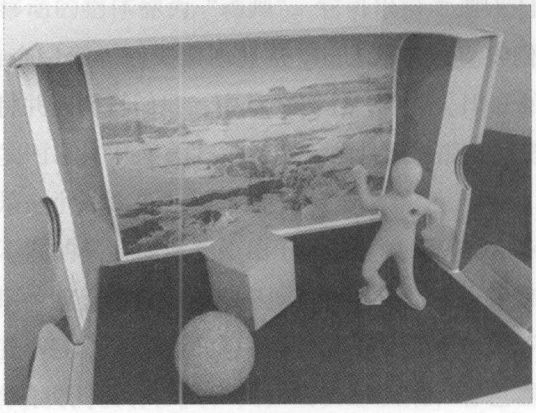
我们生活在一个令人激动无比的年代，因为你正在阅读此书，所以你也是其中之一。今后发生的任何事情都直接取决于你们。就像个人电脑的先驱艾伦·凯所说，“预测未来的方法就是去创造它”。

让我们赶快行动起来吧。

在第 2 章中，我们将直接进入 Unity 并且创建我们的第一个 3D 场景，同时学习有关世界坐标系和缩放的知识。然后，在第 3 章中，我们将创建并将其运行在 VR 头盔上，同时我们将讨论虚拟现实是如何运行的。

Chapter 2

第 2 章
物体和缩放比例



你应该记得幼年时在学校用鞋盒制作透视图的项目吧，现在让我们用 Unity 来做这个项目。让我们用几个简单的几何物体来组装第一个场景，在这个过程中，我们会谈及很多关于**空间比例尺**的内容。在本章中，我们将讨论以下话题：

- 简单地介绍一下 Unity 5 的 3D 游戏引擎。
- 用 Unity 创建一个简单的透视图。
- 制作一些测量工具，其中包括一个单位立方体和一个网格投影。
- 使用 Blender 创建一个带有纹理贴图的立方体，然后把它导入 Unity。

2.1 开始使用 Unity

如果你的电脑上还没有安装 Unity 3D 游戏引擎程序，赶紧装一个吧！全功能的个

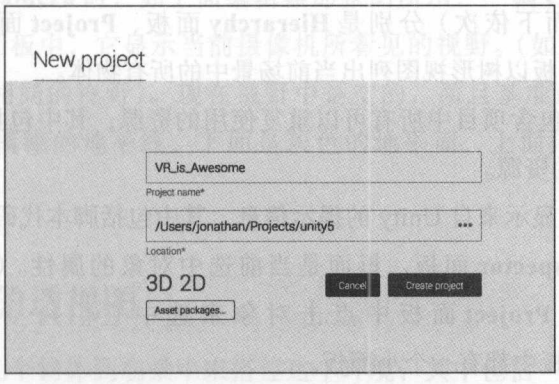
人版是免费的，而且在 Windows 和 Mac 上都能运行。你可以在 https://unity3d.com/get-unity/ 下载 Unity，选择想要下载的版本，点击“**下载安装程序**”，然后按照说明操作。本书假设读者使用的是 Unity 5.1 版本。

考虑到入门读者，相比后面的章节，第一节会慢慢来，我将手把手地教你。而且，即使你已经了解 Unity 并且开发过自己的游戏，也还是值得花点时间回顾一下这些基础概念，因为虚拟现实的玩法有时候和游戏还不太一样。

2.1.1　新建 Unity 项目

创建一个新的 Unity 项目，命名为“VR_is_Awesome”，或者起一个自己喜欢的名称。

要在 Unity 中创建一个新的项目，先从操作系统中启动 Unity，然后会出现一个 **Open** 对话框，在这个对话框中，选择 **New Project** 会打开一个 **New project** 的对话框，如下图所示：

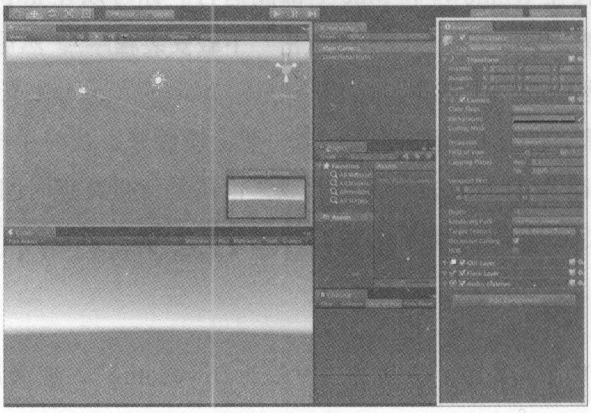


填写项目名称，并检查一下项目所在的文件夹位置是否如你所愿，确认 **3D** 选项（左下）呈选中状态。现在还不需要选择额外的资源包（asset packages），在后面需要的时候再说。点击 **Create project**。

2.1.2　Unity 编辑器

你的新项目会在 Unity 编辑器中打开，如下图所示（我把各个窗口面板的布局调整成这样是为了便于讨论和突出显示几个面板）。

Unity 编辑器包括若干个不重叠的窗口或者面板组，面板组可能会被划分成几个面板。下面是对于上图中调整过布局后的每个面板的简要说明（你自己的布局可能与之不同）。



- 左上的 **Scene** 面板用于可视化地搭建当前场景中的 3D 空间，其中包含物体的摆放。
- **Scene** 面板下方（左下）的 **Game** 视图用于显示真实游戏中的摄像机视野（现在是一片天空环绕着的空地）。在 **Play Mode** 下，游戏会运行在这个面板中。
- 中间（自上而下依次）分别是 **Hierarchy** 面板、**Project** 面板和 **Console** 面板，**Hierarchy** 面板以树形视图列出当前场景中的所有物体。
- **Project** 面板包含项目中所有可以重复使用的资源，其中包括导入的资源和接下来自己创建的资源。
- **Console** 面板显示来自 Unity 的提示信息，其中包括脚本代码的警告和错误。
- 右边的是 **Inspector** 面板，里面是当前选中对象的属性。（可以通过在 **Scene**、**Hierarchy** 和 **Project** 面板中点击对象来选中它们）。对象的每个组件在 **Inspector** 面板中都有一个小面板。
- 顶部的是主菜单栏（Mac 则是在屏幕的顶部，而不是 Unity 窗口的顶部）。后面还会使用到一个包含各种控制功能的工具栏区域，其中包括一个 play（三角形图标）按钮用于启动 Play Mode。

从主菜单栏的 **Window** 菜单中，可以根据需要打开其他的面板。编辑器的用户界面也是可配置的，每个面板都可以改变位置、调整大小，以及通过拖放把面板放进选项卡。试试吧！右上方是一个 **Layout** 选择器，可以让你选择各种默认布局或保存自己喜欢的布局。

2.1.3 默认世界坐标系

一个默认的空 Unity 场景包括一个 **Main Camera** 组件和一个 **Directional Light** 组件，在 **Hierarchy** 面板已经列出，在 **Scene** 面板中也可以看到。**Scene** 面板还显示了一个

无限基准面网格的透视图，就像一张空白的图纸。网格平铺 x 轴（红色）和 z 轴（蓝色），而 y 轴（绿色）向上。

有一个简单的方法可记住这三条轴的颜色，即记住 R-G-B 依次对应 X-Y-Z。

Inspector 面板显示当前选中项的细节：用鼠标在 **Hierarchy** 列表或场景中选中 **Directional Light**，观察 **Inspector** 面板的每个属性以及选中项关联的各个组件，其中包括 **Transform**（变换值），对象的变换值指定其在 3D 世界坐标系中的位置、旋转和缩放比例。举个例子，位置值 a（0, 3, 0）指的是基准面中心点（$X=0$，$Z=0$）以上（Y 方向）三个单位，旋转值（50, 330, 0）意思是沿 x 轴旋转 50°，沿 y 轴旋转 330°。你会发现，可以直接用鼠标在 **Scene** 面板中改变对象的变换值。

同样的，如果你点击 **Main Camera**，位置值是（0,1 ,–10），没有旋转值。也就是说，它径直地指向前方的 Z 轴正方向。

当选择 **Main Camera** 时，如上面编辑器那张图所示，一幅 **Camera Preview** 小图会被添加进 **Scene** 面板中，它显示当前摄像机所看见的视野。（如果 **Game** 选项卡是打开着的，也会看见相同的视野）。现在视野中是空的，而且基准面还没有被渲染，但是可以分辨出一条模糊的地平线，下面是灰色的地平面，上面是蓝色的默认的环绕 **Skybox**。

2.2　创建简单的透视图

现在我们添加几个物体到场景中来搭建这个环境，其中包括一个单位立方体、一个平面、一个红色球体和一个照片背景。

2.2.1　添加立方体

我们添加第一个对象到场景中——一个单位大小的立方体。

在 **Hierarchy** 面板中，用 **Create** 菜单选择 **3D Object | Cube**，主菜单栏中 **GameObject** 的下拉菜单中也有同样的选项。

一个默认的白色立方体就被添加到了场景中，放在了地平面的中心点（0, 0, 0）位置，没有旋转值和缩放值，你可以在 **Inspector** 面板中看到这些值。这是 **Reset** 设置值，可以在这个物体在 **Inspector** 面板中的 **Transform** 组件中找到。

Transform 的 **Reset** 值是 **Position**（0，0，0），**Rotation**（0，0，0），**Scale**（1，1，1）。

如果某些情况下你新添加的立方体不是这个值，可以手动设置成这些值，或者通过点击 **Inspector** 面板中 **Transform** 组件右上角的齿轮图标选择 **Reset** 来设置。

这个立方体在各维度上都是一个单位，我们后面会发现，在 Unity 中一个单位对应世界坐标系中的 1m，其局部中心点位于立方体的中心。

2.2.2 添加平面

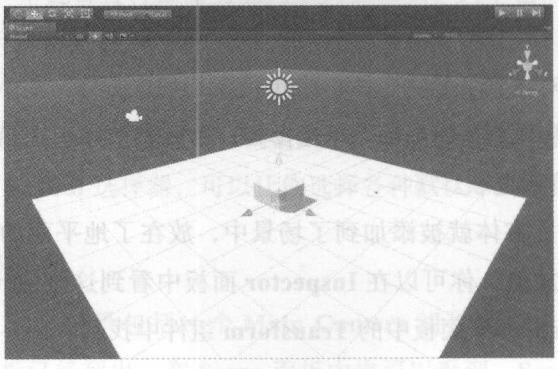
现在我们来添加一个平面对象到场景中。

在 **Hierarchy** 面板中，点击 **Create** 菜单（或 **GameObject** 菜单），然后选择 **3D Object | Plane**。

一个默认的白色平面会被添加到场景中，放置于地平面的中心点（0，0，0）位置，（如果有必要的话可以通过点击 **Inspector** 面板中 **Transform** 组件右上角的齿轮图标选择 **Reset** 来设置成这个值），把它重命名为 GroundPlane。

注意当缩放比例为（1，1，1）时，Unity 中的平面对象实际上相当于在 X 轴和 Z 轴上 10×10 个单位的长度，也就是说，GroundPlane 的长宽是 10×10 个单位大小，其 **Transform** 组件的 **Scale** 值是 1。

立方体的中心点在 **Position**（0，0，0），与地平面相同。但是看起来似乎不像是这样，**Scene** 面板可能会显示成一个把 3D 场景渲染到 2D 图片上的 **Perspective** 投影，**Perspective** 变形使立方体看起来不在地平面的中心点，但实际上它在中心点，数一下到各边的网格线数就能看出来。另外，当在虚拟现实中查看它的时候，你实际上是站在场景中的，这个时候根本看不出变形，如下图所示：



立方体陷入了地平面之下是因为其局部原点在其几何中心—也就是相当于 1×1×1 的中心点（0.5, 0.5, 0.5），这听起来可能很显然，但有些模型的原点并不在其几何中心上（比如在其一个角上）。一个对象的 **Transform** 组件的位置值是其局部原点在世界坐标系中的位置值。我们移动一下这个立方体：

1. 通过 **Inspector** 面板中把 Position 的 *Y* 值设置成 0.5：**Position**（0, 0.5, 0），把立方体移动到地平面的表面以上。

2. 通过在 *Y* 旋转值中输入 20：**Rotation**（0, 20, 0），让立方体绕着 *Y* 轴稍微旋转一点。

注意，其旋转的方向是顺时针 20°。拿出左手，比划一个竖起大拇指的手势，看看其余四个手指指向什么方向？ Unity 使用左手坐标系统（对于左手还是右手系统并没有标准，有些软件是左手坐标系统，有些是右手坐标系统）。

Unity 使用左手坐标系统，*y* 轴向上。

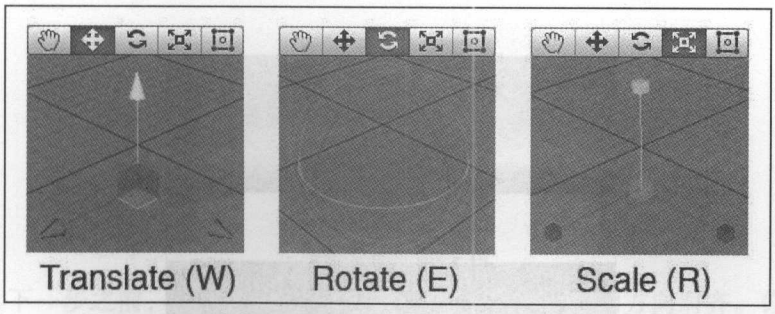
2.2.3　添加球体和材质

接下来，我们添加一个球体。

从菜单中选择 **GameObject | 3D Object | Sphere**。

像那个立方体一样，球体的半径是 1.0，原点也在几何中心。（如果有必要的话可以通过点击 **Inspector** 面板中 **Transform** 组件右上角的齿轮图标选择 **Reset** 来设置默认值）。很难看到这个球体，因为它被嵌入在立方体中，我们需要移动球体的位置。

这次，我们使用 **Scene** 面板的 **Gizmos** 组件移动这个球体，在 **Scene** 视图中，你可以选择使用图形化的控制方式或 gizmos 来改变球体的变换值，如下图所示，来自于 Unity 文档（http://docs.unity3d.com/Manual/PositioningGameObjects.html）：



在 **Scene** 面板中，选中球体，确认 **Translate** 工具处于激活状态（左上方工具栏的图标栏中的第二个图标），使用 *x* 轴、*y* 轴、*z* 轴箭头来移动它，我把**位置**变成了（1.6，0.75，-1.75）。

gizmo 是一个图形化控件，用于操作某个对象或视图的参数。Gizmos 中有拖拽点或操作点，可以用鼠标点击和拖动。

在继续下一步之前，先按如下操作保存之前的操作：

1. 在主菜单中，选择 **File | Save Scene** 然后将其命名为 Diorama。

2. 然后，选择 **File | Save Project**，注意在 **Project** 面板中，新的场景对象已经被保存在 **Assets** 文件夹的根目录中。

我们再制作一些有颜色的纹理，把纹理应用到物体上，来给场景上点颜色，步骤如下：

1. 在 **Project** 面板中，选择 **Assets** 文件夹的根目录，再选择 **Create | Folder**，重命名文件夹为 Materials。

2. 选中 Materials 文件夹，再选择 **Create | Material**，重命名为 Red。

3. 在 **Inspector** 面板中，点击 **Albedo** 右边的白色矩形会打开一个 **Color** 面板，选择一种好看的红色。

4. 再用同样的操作制作一个蓝色的材质。

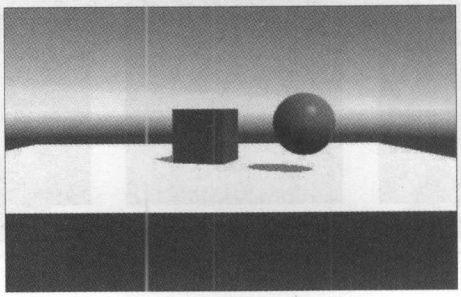
5. 在 **Hierarchy**（或 **Scene**）面板中选中 **Sphere**。

6. 将红色材质从 **Project** 面板拖进 **Inspector** 面板作为球体的材质。球体将会变红。

7. 在 **Hierarchy**（或 **Scene**）面板中选中 **Cube**。

8. 这次，将蓝色材质从 **Project** 面板拖进场景中作为立方体的材质，球体将会变蓝。

保存场景，并且保存项目。下面是场景现在的样子（和你制作的可能不太一样，不过无所谓）：



注意，我们使用的文件夹在 Project/Assets/ 目录下，用来存放我们的材料。

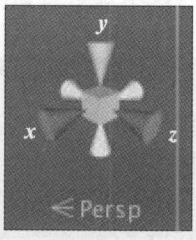
2.2.4　改变场景视图

你可以随时用各种方式改变场景视图，这取决于你的触摸板是 3 个键的还是 2 个键的，抑或只有 1 个键的 Mac。好好看一下 Unity 手册，可以在 http://docs.unity3d.com/Manual/SceneViewNavigation.html 这个链接找到你需要的文档。

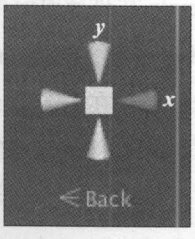
一般来说，鼠标左或右键与 Shift、Ctrl、Alt 键的组合可以执行以下操作：

- 拖动摄像机。
- 让摄像机绕着当前中心点旋转。
- 放大和缩小。
- Alt + 鼠标右键可以上、下、左、右旋转当前的视角。
- 当选中 **Hand** 工具（在图标栏的左上方）时，鼠标右键移动视野，鼠标中键也一样。

在 **Scene** 面板的右上方是 **Scene View** 小部件，把场景视图的朝向描绘成下图这样，表示的是：比如在 **Perspective** 视图中，*X* 轴延伸至左边，*Z* 轴延伸至右边。



可以通过点击相应的色锥来改变视图的方向，使我们可以直接正面观察任意一个轴。点击中间的小立方体可以将 **Perspective** 视图变成 **Orthographic**（无变形）视图，如下图所示：



在继续下一步之前，我们先把场景视图与 **Main Camera** 的方向对齐。你可能会想起

我说过的默认摄像机朝向：(0, 0, 0)，是向下看 Z 轴的正方向（从后向前）。步骤如下：

1. 点击 **Scene** 视图小部件中的红色 X 锥，把视图从 **Back（后向）** 调整为前向。

2. 再使用手形工具（或者鼠标中键）慢慢滑动视图。

现在，当选择 **Main Camera** 组件（在 **Hierarchy** 面板中）时，可以看到 **Scene** 视图大致上与 **Camera Preview** 朝向相同。（参考下一节的屏幕截图，在我们添加照片之后可以看到场景与预览看起来差不多）。

2.2.5 添加照片

现在，我们添加一张照片作为我们透视图的背景。

在计算机图形学中，映射到物体上的图片叫作纹理。当物体在世界坐标系中以 X, Y, Z 表示时，纹理以 U, V 坐标表示（与像素一样）。我们会发现纹理和 UV 贴图存在缩放的问题，步骤如下：

1. 通过菜单 **GameObject | 3D Object | Plane** 创建一个平面，命名为 PhotoPlane。

2. 重置这个平面的变换值。在 **Inspector** 面板中，找到 **Transform** 面板右上方的齿轮，点击图标选择 **Reset**。

3. 绕着 Z 轴旋转 90°（把 **Transform** 组件的 **Rotation** 的 ***Z*** 值设置成 -90）。这是负 90，所以它是竖直的，垂直于地平面。

4. 顺着 Y 轴方向旋转 90°，这样它就面向我们了。

5. 将其移动到地平面的最后面，使其 **Position** 的 ***Z***＝5，***Y***＝5（想想地平面的 10×10 个单位）。

6. 使用 Windows 资源管理器或 Mac 的 Finder 选择任意一张计算机中的图片，粘贴到这个 PhotoPlane 上。（也可以用本书中的 Grand Canyon.png 图片。）

7. 在 **Project** 面板中，选择 **Assets** 文件夹的根目录，选择菜单中的 **Create | Folder**，重命名为 Textures。

8. 拖动图片文件到 Assets/Materials 文件夹中，它应该会自动以纹理对象导入。或者，也可以在 **Assets** 文件夹上点击鼠标右键，选择 **Import New Asset...** 导入图片。

选择 **Project** 面板中的图片 **Textures**，在 **Inspector** 面板中检查其设置，如果原图为矩形，纹理会变成正方形的（比如 2 048×2 048），看起来像压扁了。当把它映射到正方形的面上时也会变扁，步骤如下：

1. 从 **Project** 面板中拖动图片到 photo plane（**Scene** 面板中）上。

哎呀！我这张图片向侧面旋转了。(你的呢？)

2. 选中 PhotoPlane，把 **Transform** 组件的 **Rotation** 的 ***X*** 值设置成 90° 。

好了，它现在竖直了，但还是扁的。我们来修复它。检查图片的原始分辨率，看它的宽高比，我的 Grand Canyon 图片是 2 576×1 932，当你用高除宽时，为 0.75。

3. 在 Unity 中，设置 PhotoPlane 的 **Transform** 组件 **Scale** 的 ***Z*** 值为 0.75。

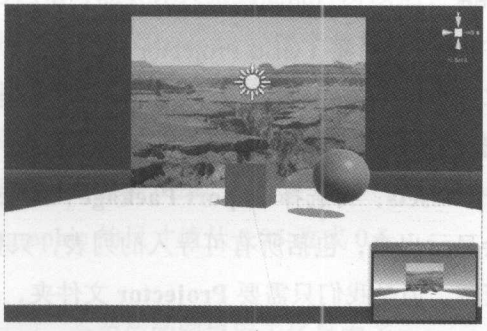
4. 把 **Position** 的 ***Y*** 值设置为 3.75。

为什么是 3.75？高是从 10 开始的。所以，我们缩放到 7.5。物体的缩放比与其原始大小有关。所以，高的一半是 3.75。我们想让 photo plane 的中心点在地平面以上 3.75 个单位。

我们已经设置了大小和位置，但是图片看起来褪色了，这是因为场景中模糊的光影响了它。你可能想保持这种效果，尤其是当你建构复杂的光照模型和材质时，但是现在，我们要把光去掉。

选中 PhotoPlane，注意照片的 **Texture** 组件在 **Inspector** 面板中有默认的 **Shader** 组件的 **Standard**，将它改成 **Unlit | Texture**。

现在我的看起来是这样了，你的应该也差不多：



看起来还不错吧，记得保存场景和项目。

2.2.6　给地平面着色

如果想改变地平面的颜色，可以创建一个新的材质（在 **Project** 面板中），命名为 Ground 并拖到地平面上。然后，改变其 **Albedo** 色。建议使用滴管（图标）从 photo plane 中拾取一种土色。

2.3 测量工具

我们已经创建了一个 Unity 场景，添加了一些基础 3D 物体，并且创建了一些基本的纹理，其中包括一张照片。然后，我们学习在 Unity 的 3D 世界空间中移动和变换对象。问题是场景中物体的实际大小并不是一直都那么明显，应该放大它们，或者用 **Perspective** 和 **Orthographic** 视图来对比，或用其他功能来影响其外观尺寸。我们来看看处理其比例的几种方式。

2.3.1 随手保留一个单位立方体

我建议随手保留一个单位立方体在 **Hierarchy** 面板中，在不需要的时候禁用它（反选 **Inspector** 面板左上边的复选框）就行。它可以当成一个测量尺或者测量仪用。我用它来预估对象的实际尺寸、对象间的距离、高度和海拔等。

创建一个单位立方体，命名为 Unit Cube，随便放在一个不碍事的位置，比如 **Position**（–2, 0.5, –2）。

还可以添加一个刻度尺来测量，让纹理精确到其边缘。

暂时把它设置为可用状态。

2.3.2 使用网格投影器

我想向你们介绍一下**网格投影器**（Grid Projector），它是一个很方便的工具，用于在任何 Unity 场景中可视化缩放。它是 **Effects** 包中 **Standard Assets** 的其中之一，所以需要把它导入项目中，步骤如下：

1. 在主菜单栏中选择 **Assets**，再选择 **Import Package | Effects**。

2. **Import** 对话框会显示出来，包括所有可导入的列表，只要你愿意，可以点击 **All** 选择 **Import** 导入所有资源，但是我们只需要 **Projector** 文件夹，所以把它导入即可。

现在添加一个 **Projector** 到场景中，步骤如下：

1. 在 **Project** 面板中找到 Grid Projector 预制件，定位到 Assets/Standard Assets/Effects/Projectors/Prefabs 文件夹。

2. 拖动一份 Grid Projector 的复本到场景中，把 **Position** 的 ***Y*** 值设置成 5，让它在地平面之上。

默认的 Grid Projector 面朝下（**Rotation** 的 ***X*** = 90），这也通常是我们所期待的。在 **Scene** 视图中，可以看到正交投影射线。有一篇 Unity 文档（http://docs.unity3d.com/

Manual/class-Projector.html）是这样解释的：

投影器可以把材质投影到所有与其视锥体相交的物体上。

意思是说，与投影射线相交的物体都会得到投影后的材质。

在这个例子中，如你所愿，投影器的材质（名字也叫 GridProjector）有一个“Grid”纹理，它看起来简直像一个十字准星（Assets/.../Projectors/Textures/Grid，你自己看）。

默认情况下，投影器像一束光一样照耀在网格图案表面。在我们的场景中，GroundPlane 平面是明亮的，所以网格不会显示出来。现在我们按如下步骤操作：

在 **Hierarchy** 面板中选中网格投影器，把 GridProjector 材质组件放进 **Inspector** 面板，再把它的 **Shader** 从 **Hidden/Projector Light** 改成 **Hidden/Projector Multiply**。

它现在会在一片黑色上绘制白色网格线，要想得到更好的效果，把场景视图改成 **Top** 的视图朝向，可按如下操作：

1. 点击 **View** 面板中场景视图小部件中右上方的绿色的 **Y** 锥形。

2. 点击小部件中的小立方体，把 **Perspective** 变成 **Orthographic**（无变形）视图。

现在你可以从上向下看到地平面上，选中网格投影器（确认左上方工具栏的第二个图标平移工具是激活状态），慢慢用小部件的平移工具移动投影器，网格线也随之移动，你可以把它放在 **Position**（–2.5，5，–0.5）的位置上以避免投影器挡着光线。

现在这个内置的参考网格可能会让人感到混乱，所以把它关了吧：

1. 在 **Scene** 视图面板中，点击 **Gizmos**（找这个名字的菜单，有控制你的所有小部件的选项），反选 **Show Grid**。

好了，看到了吧？默认的网格尺寸是单位立方体的边长的一半。在 **Inspector** 中，投影器组件的 Orthographic 的尺寸值是 0.25。

2. 将投影器中 Orthographic 的尺寸值从 0.25 改成 0.5。

3. 保存场景和项目。

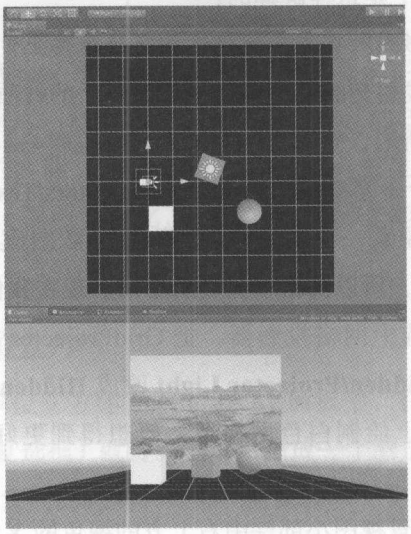
现在我们就可以随时把一个单位的网格用于场景中了。

让我们保留这个状态一会儿，因为它看起来挺酷的，如下图所示：

2.3.3　测量 Ethan 角色

一个虚拟角色有多大？Unity 中有一个叫作 Ethan 的第三人称角色，我们把他添加到场景中，他是 **Characters** 包的 **Standard Assets** 之一，所以我们需要将其导入项目。

按如下步骤操作：



1. 在主菜单栏中选择 **Assets**，然后选择 **Package | Characters**。

2. 在弹出的 **Import** 对话框中，有一个可导入列表。点击 **All** 再点击 **Import**。**ThirdPerson-Controller** 是 **Project** 面板中的一个预制件（预置资源），可以在 Assets/Standard Assets/Characters/ThirdPersonCharacter/Prefabs 文件夹中找到。

3. 拖一个 **ThirdPersonController** 的复本到场景中，*X* 和 *Z* 的位置值无所谓，但是要把 *Y* 值设置成 0，这样名字叫作 Ethan 的角色就站在地面上了，我设置的坐标值为（2.2, 0, 0.75）。

我们试试效果：

1. 点击在 Unity 窗口顶部中间的 Play 图标运行游戏。使用 W、A、S、D 键来移动。跑，Ethan! 跑!

2. 再次点击 Play 图标来停止游戏，进入编辑模式。

所以，Ethan 有多高？根据 Google 搜索的结果，人类中男性的平均身高是 1.68m（在美国成年男性平均身高是 1.77m），我们来看看 Ethan 有多高：

- 使用平移工具小部件滑动单位立方体到接近 Ethan 处。看出来了，他大概身高是单位立方体高度的 1.6 倍。
- 调整立方体的高度值（***Y***）到 1.6，再把位置值 *Y* 调整到 0.8。

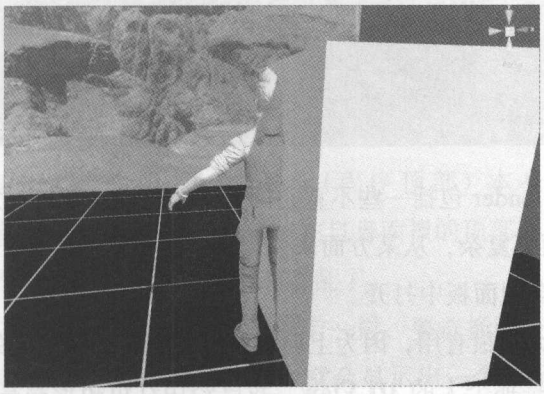
我们再来看看，如下图所示，他身高数值不到 1.6，所以 Ethan 比平均身高要矮一点（除非你算上他的发尖）。滑动视图，我看到的是 Ethan 的右脸，然后再调整立方体的视平

线大约是 1.4m。记录一下：

1. 把单位立方体恢复到 **Scale**（1, 1, 1）和 **Position**（–2, 0.5, –2）。

2. 保存场景和项目。

下图是 1.6 个单位高度的立方体与 Ethan 的比较：



2.4　从 Blender 实验中导入

Unity 提供了一些基本的几何形状，但是如果涉及更复杂的模型，就需要使用 Unity 之外的工具了。**Unity Store** 中有海量优质模型（以及很多其他类型的资源），也有其他很多网站的社区共享 3D 模型，有收费的也有出于兴趣的。这些模型是怎么来的？我们导入模型时会遇到问题吗？

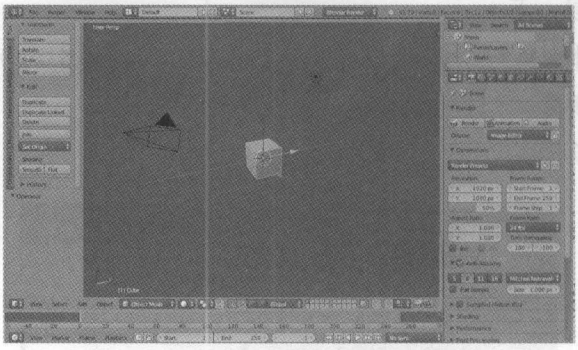
本书虽然是关于 Unity 的图书，但我要打一个擦边球，这里我要使用 Blender（2.7x 版本），一个免费开源的 3D 动画套装（http://www.blender.org/），制作模型并导入 Unity。喝一杯咖啡开始享受它吧。

现在并不是要制作什么高级的东西，而只是制作一个立方体和一个简单的纹理贴图。这个练习的目的是要弄明白如何把一个单位立方体从 Blender 以同样的比例和朝向导入 Unity。

你也可以跳过本节，或者试试其他你喜欢的建模软件。如果你不想遇到麻烦，可以下载随书的打包文件找到为本节创建的文件。

2.5　Blender 简介

打开 Blender 程序，关闭欢迎界面后呈现的就是 Blender 编辑器了，与下面的截图差不多：



同 Unity 一样，Blender 包括一些不重叠的窗口，可以按喜好自定义这些窗口的布局。但是，Blender 的界面更复杂，从某方面来说因为它集成了若干个不同的编辑器，而这些编辑器可以同时在各自的面板中打开。

改变默认视图的大小很有用，因为上面那张图中包含 5 个不同的编辑器。

最明显的编辑器是那个大的 **3D View**，我已经用红色矩形高亮显示了，在这个视图中可以察看、移动和组织 Blender 场景中的对象。

其余 4 个打开的编辑器是：

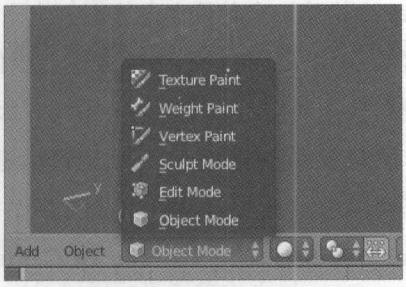
- **Info editor**，会发现它沿着程序顶部的边缘，其中包含程序的全局菜单和信息。
- **Timeline editor**，在程序的下底边上，用于动画。
- **Outliner editor**，在右上方，场景中所有对象的层级视图都在这里。
- **Properties editor**，在 **Outliner** 的右下方，是一个很有用的面板，可以察看和修改场景中对象的多个属性。

每个编辑器也可以有多个面板，我们来看看 3D 视图编辑器：

- 中间的大片区域是 **3D Viewport**，可以察看、移动和组织 Blender 场景中的对象。
- 3D Viewport 的下方是编辑器的 **Header**，它虽然在下方我们却叫它 Header。这个 Header 是一行菜单和工具栏，提供了很多控制编辑器的功能，包括选择器、编辑模式、变换操作和层次管理。
- 左边的是 **Tool Shelf**，包括各种编辑工具，可以用于当前选中对象、放进选项卡，**Tool Shelf** 可以通过滑动其边缘或者按 *T* 键来切换开关状态。
- 3D 视角也有一个 **Properties** 面板，它默认可能是隐藏的，可以按 N 键来切换开关状态，它提供了当前选中对象的属性设置。

在接下来的介绍中，会要求你改变 3D 视图编辑器的 **Interaction Mode**，变成 **Edit**

Mode 和 **Texture Paint** 模式之间的值，可以在 Header 中选择，如下图所示：



其他编辑器也有 Header 面板，Info editor（程序顶部）本身就是一个 Header！Outliner 和 Properties 编辑器（右边）的 Header 在自身面板的顶部而不是在底部。

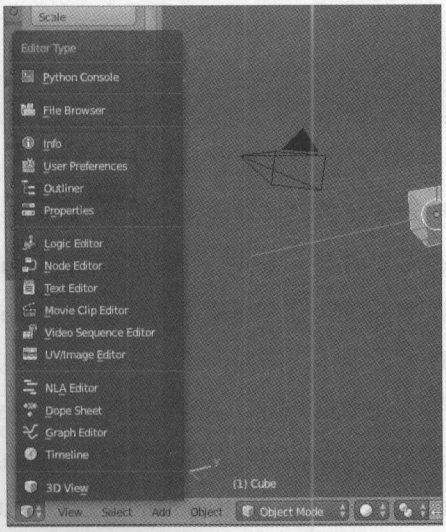
调整过布局后，看起来就不是那么拥挤和混乱了。

Properties 编辑器有一长溜图标，像是选项卡一样，要选择面板中的其余属性。鼠标悬停在图标上（这里其他的 UI 部件也都一样）就会显示出一个提示语告诉你它的功能，后面的（后续章节中用到它的时候）图片中可以看到。

Blender 的布局非常灵活，你甚至可以把一个面板从一个编辑器变成另外一个，每个 Header 的最左边是 **Editor Type** 选择器。点击它就可以看到所有选项。

除了 Blender 界面上可以点击东西之外，还可以用键盘快捷键执行命令，如果忘了在哪找到某个功能，可以敲空格键再输入最接近你要找的命令名称，它可能就会弹出来！

下图是 Blender 中的 **Editor Type** 选择器：



2.5.1 立方体

现在，我们来在 Blender 中构建一个单位立方体。

默认的场景中可能已经有物体了，包括一个立方体、摄像机和一个光源，如同前面在 Blender 窗口中显示的那样。(你的初始设置可能会不同，因为它是可配置的。)

如果初始场景中没有单位立方体，那么就像下面这样创建一个：

1. 删除场景中的任何东西以确保场景是空的（右键选择，X 键删除）。

2. 用 Shift＋S（打开 **Snap** 选项列表）| **Cursor To Center** 把 3D 光标的原点设置成 (0, 0, 0)。

3. 在左边的 **Tool Shelf** 面板中，选择 **Create** 选项卡，再在 **Mesh** 下选择 **Cube** 以添加一个立方体。

好了，我们现在进度一样了。

注意在 Blender 中，参考网格延伸了 x 轴和 y 轴，z 轴朝上（不像 Unity，y 轴朝上）。

还有，注意 Blender 中默认的立方体的大小是 (2, 2, 2)。

而我们需要的是一个单位立方体立在基准面的原点之上，就像这样操作吧：

1. 用 N 键打开 Properties 面板。

2. 选择 **Transform | Scale** 把 ***X***，***Y***，***Z*** 设置成 (0.5, 0.5, 0.5)。

3. 选择 **Transform | Location** 把 ***Z*** 设置成 0.5。

4. 再按 N 键隐藏面板。

5. 可以用鼠标滚轮来缩放。

我们还要确保当前渲染器是 **Blender Render**（在程序窗口顶部中间的 Info 编辑器中的下拉选项里）。

2.5.2 UV 纹理图片

接下来，我们创建一张 UV 纹理图片：

1. 找到 **Edit Mode**，在底部的 Header 栏中选择 **Interaction Mode** 选择器。

2. 用全选（双击 A 键）确保所有表面都被选中。

3. 在左边的 **Tool Shelf** 面板中，选择 **Shading/UVs** 选项卡。

4. 在 **UV Mapping** 下方点击 **Unwrap**，在下拉列表中选择 **Smart UV Project**，接受默认值，点击 **OK**（结果见下图，也能看到光皮的立方体长什么样）。

5. 现在再用底部的 Header 栏把 **Interaction Mode** 变成 **Texture Paint** 模式。

6. 我们还需要为我们的材质定义一个绘制槽，点击 **Add Paint Slot**，选择 **Diffuse Color**，命名为 CubeFaces，点击 **OK**。

我们现在开始直接在立方体上绘制，首先画前面，步骤如下：

1. 制作一个小一点的画刷。在左边的 **Tool Shelf** 面板中，**Tools** 选项卡下选择 **Brush | Radius**，然后输入 8 px。

2. 可能在正交视图下更容易操作。在底部菜单栏中选择 **View | View Persp/Ortho**。

3. 然后选择 **View | Front**。

4. 如果需要的话你可以用鼠标滚轮来缩放大小。

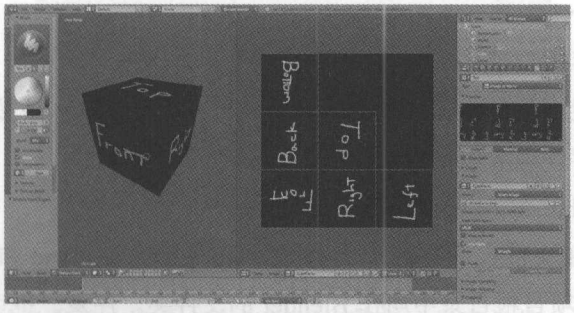
5. 用你最好的书法水平用鼠标左键写上单词“Front”。

6. 再在背面做同样的操作。

7. 在底部的菜单栏中选择 **View | Back** 再用右键选择这一面。

8. 好好写一个“Back”。

再在左、右、上、下的面上重复上面的操作，如果发生不能画的情况，确认是否选中了当前表面，试试用右键在这个面上重新选择，结果应该差不多像下面这样（同时在 **3D View** 编辑器中用正交透视和 **UV/Image Editor**）：



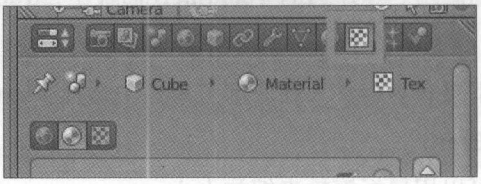
现在，我们保存图片，然后设置其属性，步骤如下：

1. 使用 **3D View** 编辑器底部 Header 的最左边的选择器，把当前的 **Editor Type** 改成 **UV/Image Editor**。

2. 点击 **Browse Image to be linked** 选择器图标。（朝着＋图标的左边），再在列表中选择 CubeFaces。

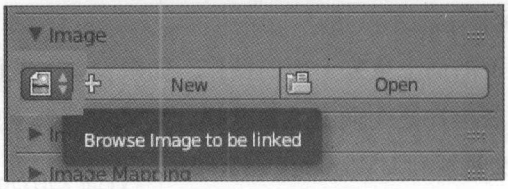
3. 底部菜单栏的 **Image** 菜单项现在有一个星号（**Image***）表示有一张未保存的图片，点击它，选择 **Save As Image**，保存成 CubeFaces.png，使用一个 Unity 工程之外的文件夹。

4. 在左边，在 Properties 编辑器的面板上，在其 Header 中找到一长排图标，选择 **Texture**（倒数第三个），如果面板不够宽的话它可能会不显示，你可以用鼠标下滑来显示，如下图所示：



5. 在 **Texture** 属性中，把 **Type** 变成 **Image or Movie**。

6. 在属性的 **Image** 组中，点击 **Browse Image to be linked** 选择器图标（见下图）选择 CubeFaces。



7. 你应该在 **Preview** 窗口中可以看到带有标签的纹理图片了。

不错！我们保存这个 Blender 模型吧，步骤如下：

1. 在顶部菜单栏中的 Info 编辑器中选择 **File**，再点击 **Save**（或按 Ctrl+S）。

2. 使用之前保存纹理图片的文件夹。

3. 命名为 UprightCube.blend，点击 **Save Blender File**。

我们现在应该在文件夹中有两个文件了，UprightCube.blend 和 CubeFaces.png。我使用了一个在 Unity 项目根目录中叫作 Blender 的文件夹。注意，也可以导出成其他标准格式，比如 **FBX**（Filmbox 的简称）就可作为一种选择。

太棒了，完成这么多了。没弄懂也没关系，Blender 可能比较难，但是 Unity 需要模型，你可以一直都从 Unity 的 **Asset Store** 和 3D 模型分享网站中下载其他人的模型。但不要做一个没用的人，学习自己做自己的模型吧。哈哈！我说真的，这是一个很好的学习起点。

2.5.3 导入 Unity

回到 Unity，我们要导入刚才那两个文件——UprightCube.blend 和 CubeFaces.png，

一个一个来，步骤如下：

1. 在 **Project** 面板中，选择 **Assets** 根目录，定位到 **Create | Folder**，重命名这个文件夹为 Models。

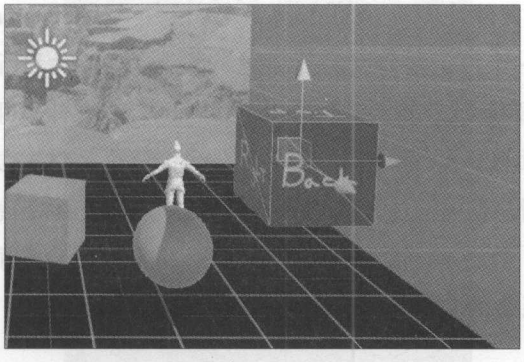
2. 有一个简单的方法导入文件到 Unity 中，就是从 Windows 资源管理器（或 Mac 的 Finder）窗口拖拽 .blend 文件到 **Project** 面板的 **Assets/Models** 文件夹，把 .png 文件拖拽到 **Assets/Textures** 文件夹中（或者也可以用主菜单栏中的 **Assets | Import New Assets**）。

3. 通过从 **Assets/Models** 文件夹，也就是刚才导入 **Scene** 视图的地方，把刚才导入的模型 UprightCube 拖动到场景中。

4. 把它的坐标值设置成与其他对象远离，我设置的 **Position** 值是（2.6, 2.2, –3）。

5. 从 **Assets/Textures** 文件夹中拖动 CubeFaces 纹理到 **Scene** 视图中，停在刚才添加的 UprightCube 上让它接收这个纹理，然后把松开纹理把放到立方体上。

这样场景应该看着像下面这样了：



2.5.4　观察者

立方体的背面对着我们，这是一个失误吗？其实这是正常的，因为当前视点是朝前看的，那么我们就应该看着立方体的背面。你应该没注意到，Ethan 也是这样。而这个立方体看着差不多有一个单位的容积。

但是，仔细检查一下，在立方体的 **Inspector** 面板中，会发现它导入的缩放比例是我们在 Blender 中给的（0.5, 0.5, 0.5），另外它还有一个 *X* 轴的 –90° 旋转（负 90），所以如果我们重置变换值，也就是比例尺为（1, 1, 1），它在我们的世界坐标系中将会是两个单位大小，而且是颠倒的（所以，别重置）。

不用回到 Blender 中，稍微操作一下就可以抵消这个旋转值了。

Blender的默认朝上的方向是*Z*轴，而Unity是*Y*轴。所以，导入时*X*轴-90°的旋转用于调整这个差异。导入的比例可以在对象的**Inspector**面板的**Import Settings**中调整。

比例问题可以通过下面的步骤修复：

1. 在**Project**面板中，选择我们导入的UprightCube。**Inspector**面板会显示其**Import Settings**。

2. 把**Scale Factor**的值从1改成0.5。

3. 点击**Apply**。

场景中的立方体现在可以是**Scale**（1, 1, 1），且可以是一个单位立方体，如我们所愿。

4. 在**Hierarchy**中选择UprightCube，把**Transform**的**Sacle**值改成（1, 1, 1）。

在结束上面这个过程之前，在**Hierarchy**面板中选择UprightCube并把它拖进**Project**面板的**Assets**文件夹中。（你可以考虑建立一个**Assets/Prefabs**文件夹把文件放进去。）这样就做成了一个可重复使用的预制件，包括纹理图片和所有内容。

本节中的一些重要练习（除了在Blender学习的那些之外）可以用于任何3D Unity项目，包括虚拟现实项目。正常来说，你应该会导入比立方体要复杂得多的模型，会遇到数据转换、缩放比例、朝向、UV纹理图片等相关的问题让你困惑。如果遇到了，试着把问题分解成小问题，分解成更独立的场景。做一些小的试验来了解程序是如何交换数据的，这对于你理解混杂的参数可能会有所帮助。

小结

本章中，我们构建了一个简单的透视图，也熟悉了Unity的编辑器，还学习了在设计场景时世界比例尺的重要性，我们还设置了一些游戏中的工具来帮助我们在后续章节中处理比例和定位。最后，我们进入了Blender，用UV纹理构建了一个模型，并且了解了一些把模型导入到Unity时可能遇到的问题。

下一章中，我们将用虚拟现实摄像机设置一个项目，然后构建并在你的VR头盔中运行。

第 3 章 Chapter 3

虚拟现实的构建和运行

很好，非常酷，但是我的虚拟现实在哪里？我要我的虚拟现实！

别急，孩子，马上就有。

本章中，我们将建立一个可以构建并运行于虚拟现实头盔显示器之中的项目，然后会详细讨论虚拟现实硬件技术的运行原理，将涉及下列话题：

- 虚拟现实设备集成软件的范围。
- 为虚拟现实设备安装和构建一个项目。
- 用于虚拟现实技术运行原理的细节和定义术语。

本书中的项目不需要按顺序实现，可以自由地跳章阅读，因为后一章并不依赖于前一章。然而本章例外，请在阅读余下章节之前先实现 MeMyselfEye 预制件和 Clicker 类。

3.1 虚拟现实设备集成的软件

在深入讨论之前，我们先了解几种将我们的 Unity 项目集成进虚拟现实设备的方式。一般来说，Unity 项目必须包含一个摄像机对象，用于渲染两套立体视图，在虚拟现实头盔中为眼睛提供视图。

用于虚拟现实硬件中集成程序的软件范围很广，从内置的支持软件和设备特有的接口到不依赖于设备和平台的软件。

3.1.1 Unity 对虚拟现实的内置支持

Unity 从 5.1 开始，已经内置了对虚拟现实头盔的支持。写本书之时，它已经可以直接支持 Oculus Rift 和三星的 Gear VR（由 Oculus 的软件驱动），对其他设备的支持也已经宣布了，其中包括索尼的 PlayStation Morpheus。你可以使用标准的摄像机组件，比如附加到 **Main Camera** 的和标准的人物角色资源预制件。当你构建的项目在 **Player Settings** 中开启 **Virtual Reality Supported** 时，Unity 会将立体摄像机视图渲染并运行于头盔显示器上。

3.1.2 设备特有的 SDK

如果 Unity 没有直接支持某款设备，这个设备商将有可能发布一个 Unity 插件包。使用设备特有接口的一个好处是可以直接利用下层硬件的特性。

例如，Steam Valve 和 Google 就为 Vive 和 Cardboard 提供了设备特有的 SDK 和 Unity 包。如果你正在使用上述的其中一款设备，那么你很可能需要使用它的 SDK 和 Unity 包。（写本书之时，这些设备未被 Unity 内置的虚拟现实支持。）而对于 Oculus，Unity 5.1 中直接支持，提供了 SDK 工具包以调用其接口（参见 https://developer.oculus.com/documentation/game-engines/latest/concepts/unity-intro/）。

设备特有的软件锁定其只能构建到指定的设备上，如果这对你来说是个问题，那么你要么写点聪明的代码，要么用接下来几条途径替代。

3.1.3 开源虚拟现实项目

2015 年 1 月，**雷蛇有限公司**（Razer Inc）领导一些行业领袖宣布了**开源虚拟现实**（OSVR）平台更多内容请参见 http://www.osvr.com/），计划开发开源的硬件和软件，其中包括来自于不同厂商而可运行于多种设备的 SDK。这个开源中间件项目提供了不依赖于

设备的 SDK（和 Unity 包），这样一来你就可以在代码中使用一个接口而不用去关心用户使用什么设备了。

使用 OSVR 可以为特定的操作系统（比如 Windows、Mac 和 Linux）构建 Unity 游戏，并且可以让用户根据其使用的硬件来配置（下载之后的）应用程序。写本书之时，此项目还处于其早期阶段，快速发展中且尚不适用于本书，但是我鼓励你关注它的发展。

3.1.4　WebVR

WebVR（更多信息请参见 http://webvr.info/）是一个 JavaScript API，正在被集成进主流的网页浏览器。就像 **WebGL**（用于 Web 的 2D 和 3D 图形 API）对虚拟现实渲染和硬件的支持。现在 Unity 5 已经引入了对 WebGL 的构建支持，我相信 WebVR 的支持也会随之而来，不是 Unity 就是第三方开发者。

我们知道，浏览器几乎运行于任何平台。所以，如果你的游戏的目标平台是 WebVR，那么你不需要关心用户的操作系统，更不必关心用户使用的是什么虚拟现实硬件了！总之，就是这个意思。新的技术，比如即将到来的 **WebAssembly**，是一种新的用于 Web 平台上的二进制格式，能够帮助获取你的硬件的最佳性能，并且能够让基于 Web 的虚拟现实可行。

对于 WeVR 库，请查阅以下内容：

- **WebVR 样例**：https://github.com/borismus/webvr-boilerplate
- **GLAM**：http://tparisi.github.io/glam/
- **glTF**：http://gltf.gl/
- **MozVR**（Mozilla **Firefox Nightly** 为虚拟现实构建）：http://www.mozvr.com/
- **WebAssembly**：https://github.com/WebAssembly/design/blob/master/FAQ.md

3.1.5　3D 世界

有一些第三方的 3D 世界平台提供多用户在共享虚拟空间中的社交体验，你可以与其他玩家聊天，通过入口（portals）在不同房间中移动，甚至不需要成为专家就可以构建复杂的交互和游戏。

对于 3D 虚拟世界的案例，请查阅以下内容：

- **VRChat**：http://vrchat.net/
- **JanusVR**：http://janusvr.com/

- **AltspaceVR**：http://altvr.com/
- **High Fidelity**：https://highfidelity.com/

举个例子来说，VRChat 让你可以在 Unity 中开发 3D 空间和虚拟角色，用 VRChat 的 SDK 导出，然后加载进 VRChat，其他人就可以在社交虚拟现实体验中通过互联网分享实时。我们将在第 10 章继续探究。

3.2 创建 MeMyselfEye 预制件

开始之前，我们先创建一个对象作为虚拟环境中用户的代理，对后面会有所帮助。而且它可以简化本书中的讨论，因为不同的虚拟现实设备可能会使用不同的摄像机资源文件，就像是你的 VR 灵魂……

我们按照下面的步骤来创建这个对象：

1. 打开 Unity 并打开上一章中的项目，然后用 **File | Open Scene** 打开 **Diorama** 场景（或在 **Project** 面板中的 **Assets** 目录下双击此场景）。

2. 在主菜单栏中，点击 **GameObject | Create Empty**。

3. 把此对象重命名为 MeMyselfEye。（嘿，它就是 VR 了！）

4. 把它的坐标值设置成 **Position**（0, 1.4, –1.5），让它靠近场景。

5. 在 **Hierarchy** 面板中，把 **Main Camera** 对象拖进 MeMyselfEye 成为其子对象。

6. 选中 **Main Camera** 对象，重置其变换值（在 **Transform** 面板的右上方点击齿轮图标再选择 **Reset**）。

Game 视图应该显示成我们在场景中了，如果你回想一下之前完成的 Ethan 实验，我当时把位置 *Y* 设置成 1.4，让我们与 Ethan 在同一视平线上。

现在我们把它保存成一个可重用的预制对象，或称其为预制件，放在 **Project** 面板中的 **Assets** 目录下。我们可以在本书其他章节中的其他场景中再次使用它：

1. 在 **Project** 面板中的 **Assets** 目录下，选择 Assets 根目录，右键选择 **Create | Folder**，把文件夹重命名为 Prefabs。

2. 把 MeMyselfEye 预制件拖进 **Project** 面板的 Assets/Prefabs 文件夹下。

现在，我们来为你特定的 VR 头盔配置此项目。

我们将在本书中贯穿使用本章中的 MeMyselfEye 预制件，作为我们项目中的一个便捷通用的虚拟现实摄像机资源。

3.3　为 Oculus Rift 构建项目

如果你有 Rift，你应该已经下载了 **Oculus Runtime**、示例应用程序，以及众多优秀的游戏。要为 Rift 开发，你需要确保 Rift 在你使用 Unity 的机器上运行良好。

Unity 内置支持 Oculus Rift，你只需要按下面步骤配置你的 **Build Settings...**：

1. 在主菜单中，打开 **File | Build Settings...**。
2. 如果当前场景没有列出在 **Scenes In Build**，点击 **Add Current**。
3. 在 **Platform** 左边的列表中选择 **PC，Mac，& Linux Standalone**，然后点击 **Switch Platform**。
4. 在右边的 **Select** 列表中选择 **Target Platform OS**（比如 **Windows**）。
5. 然后，点击 **Player Settings...**；再然后，打开 **Inspector** 面板。
6. 在 **Other Settings** 下，勾选 **Virtual Reality Supported**，如果弹出 **Changing editor vr device** 对话框就点击 **Apply**。

我们测试一下，确认 Rift 连接完好并且呈打开状态。点击程序顶部中间的 Play 按钮。戴上头盔，应该会非常酷！在 Rift 中，你可以向四周看——左、上、右、下，以及你的后方，你还可以俯身或前倾。使用键盘，你可以让 Ethan 走动、跑动、跳跃，就像上一章那样。

现在我们用下面的步骤把游戏构建成一个单独的可执行的程序。与你以前的操作步骤很像，至少对于非虚拟现实应用程序基本一样：

1. 从主菜单栏中，选择 **File | Build Settings...**。
2. 点击 **Build**，并设置名称。
3. 我喜欢让程序保存在一个叫作 Builds 的子目录下，喜欢的话你也可以创建一个。
4. 点击 **Save**。

可执行程序将会被创建在 Builds 目录下，如果你使用的是 Windows，可能还会创建出一个 rift_Data 文件夹保存构建数据。像运行其他程序一样运行 Diorama——双击并选择 **Windowed** 选项，这样我们可以用屏幕右上方标准的关闭图标随时退出。

3.4　为 Google Cardboard 构建项目

如果你的 Google Cardboard 的目标平台是 Android 或 iOS，请阅读本节。

《适用于 Unity 的 Google Cardboard 入门指南》是一个很好的切入点（更多信息请访

问 https://developers.google.com/cardboard/unity/getstarted)。

3.4.1 配置 Android 环境

如果你从来没在 Android 环境上开发过，那么需要先下载和安装 Android SDK。看一下 Unity 手册中关于 Android SDK 安装的文章（http://docs.unity3d.com/Manual/android-sdksetup.html），还需要安装 Android Developer Studio（或至少要安装 SDK 工具包）和其他相关工具，比如 Java（JVM）和 USB 驱动程序。

最好先用一个不包含 Cardboard SDK 的 Unity 项目试着构建和运行，以确保环境都准备就绪（一个只有一个立方体的场景就行），确认你知道如何把程序安装和运行在你的 Android 手机上。

3.4.2 配置 iOS

Unity 手册中 *Getting started with iOS Development* 指南是一个很好的切入点（http://docs.unity3d.com/Manual/iphone-GettingStarted.html）。你只能在 Mac 上做 iOS 开发，必须配置好一个苹果开发者账号（已经支付每年的标准会员费），还需要下载和安装 Xcode 开发工具（通过 Apple Store）。

最好先用一个不包含 Cardboard SDK 的 Unity 项目试着构建和运行，以确认环境都准备就绪（一个只有一个立方体的场景就行），确保你知道如何把程序安装和运行在你的苹果手机上。

3.4.3 安装 Cardboard 的 Unity 包

要让我们的项目在 Google Cardboard 上运行，先通过这个网址下载其 SDK（https://developers.google.com/cardboard/unity/download）。

在 Unity 项目中，按以下步骤导入 CardboardSDKForUnity.unitypackage 资源包：

1. 在 **Assets** 主菜单栏中，选择 **Import Package | Custom Package...**。
2. 找到并选择 CardboardSDKForUnity.unitypackage 文件。
3. 确认所有资源都被勾选，点击 **Import**。

浏览刚才导入的资源。在 **Project** 面板中的 Assets/Cardboard 文件下中有很多有用的东西，其中包括 CardboardMain 预制件（紧挨着还有一个 CardboardHead，其中包含一个摄像机对象），在 Cardboard/Script 文件夹中还有一些脚本，可以查阅一下。

3.4.4　添加摄像机

现在，我们把 Cardboard 摄像机放进 MeMyselfEye，步骤如下：

1. 在 **Project** 面板中，找到 Assets/Cardboard/Prefabs 文件夹下的 CardboardMain。

2. 把它拖进 **Hierarchy** 面板中的 MeMyeselfEye 对象中，让它成为其子对象。

3. 在 **Hierarchy** 面板中选择 CardboardMain，看一下 **Inspector** 面板，确认 **Tap is Trigger** 选项是选中状态。

4. 在 **Hierarchy** 面板中选择 **Main Camera**（在 MeMyselfEye 内），然后通过反选 **Inspector** 面板左上方的 **Enable** 复选框禁用它。

最后，把这些修改应用到预制件上，步骤如下：

1. 在 **Hierarchy** 面板中选择 MeMyeselfEye 对象，然后在 **Inspector** 面板中 **Prefab** 旁边点击 **Apply** 按钮。

2. 保存场景。

这样，我们就把默认的 Main Camera 替换成虚拟现实摄像机了。

3.4.5　构建设置

如果你知道如何从 Unity 构建和安装程序到手机上，那么对于 Cardboard 的操作也相去无几：

1. 在主菜单栏中，选择 **File | Build Settings...**。

2. 如果当前场景没有出现在 **Scenes in Build** 的列表中，点击 **Add Current**。

3. 在左边的 **Platform** 列表中选择 **Android** 或者 **iOS**，点击 **Switch Platform**。

4. 然后，点击 **Inspector** 面板中的 **Player Settings...**。

5. 对于 **Android**，确保 **Other Settings | Virtual Reality Supported** 是**未选中**状态，因为这是用于 GearVR（通过 Oculus 驱动程序），而不是用于 Cardboard Android 的。

6. 选择 **Other Settings | PlayerSettings.bundleIdentifier**，输入一个有效的字符串，比如 com.YourName.VRisAwesome。

7. 在选项 **Resolution and Presentation | Default Orientation** 下将值设成 **Landscape Left**。

3.4.6　试玩模式

测试的话，不需要连接手机，只需要按下程序顶部中间的 Play 按钮进入 **Play Mode**（试玩模式）。可以在 **Game** 视图中看到分屏的立体视图。

在试玩模式下，你可以模拟戴着 Cardboard 头盔移动头部，用 Alt 键结合鼠标移动以平移和前后倾斜，用 Ctrl 键结合鼠标移动以左右倾斜头部。

你还可以用鼠标点击模拟磁力点击（我们会在后面关于用户输入的章节中展开讨论）。

要注意因为这是在手机上模拟运行，并没有键盘，我们之前用键盘上的键位移动 Ethan 在这里行不通。

3.4.7 构建并在 Android 中运行

要把游戏构建成一个单独的可执行的应用程序，执行以下步骤：

1. 在主菜单栏中，选择 **File | Build & Run**。

2. 设置要构建的项目名称，我喜欢把构建结果保存在一个叫 Build 的子目录中，你也可以这么做。

3. 点击 **Save**。

这样就会生成一个可运行在 Android 上的 .apk 文件，然后把这个程序安装在手机上。下面的截图显示了 Diorama 场景运行在 Android 和 Cardboard 手机中（Unity 的开发显示器在背影中）。

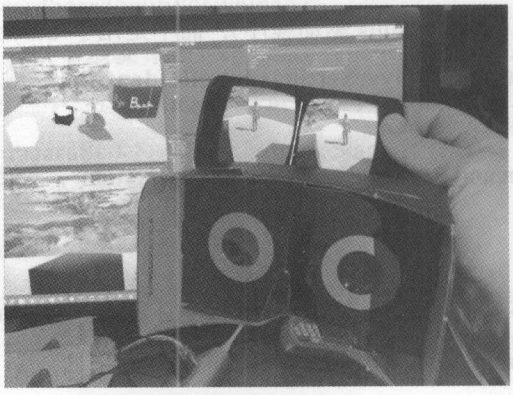
3.4.8 构建并在 iOS 中运行

要把游戏构建后运行在 iPhone 上，执行以下步骤：

1. 通过 USB 线 / 口把手机接在计算机上。

2. 在主菜单栏中，选择 **File | Build & Run**。

这样就会生成一个 Xcode 项目，运行 Xcode 并在 Xcode 中构建程序，然后把程序安装在手机上。



古董级的立体图片（链接：https://www.pinterest.com/pin/493073859173951630/）

3.5　不依赖于设备的点击器类

在这里我还需要做一件事，它对于后面的章节非常有帮助。在写本书之时，虚拟现实的输入还不能跨平台，输入设备不一定能适配 Unity 自己的 **Input Manager** 和 API。事实上，虚拟现实的输入是一个巨大的话题，值得写一本书去讨论，所以在这里就简言之。

作为对史蒂芬·乔布斯的悼念，以及对第一台苹果计算机的复古，我将限制这些项目尽量为一键输入！我们来写一个脚本用于检查键盘、鼠标以及其他设备的任何输入。（我在上一章中已经给出了一个详细的关于 Unity 脚本的介绍，所以现在请就按照步骤操作。）

1. 在 **Project** 面板中，选择 **Assets** 的根目录。
2. 点击右键并选择 **Create | Folder**，命名为 Scripts。
3. 选择 Scripts 文件夹，点击右键选择 Create | C# Script，命名为 Clicker。
4. 在 **Project** 面板中双击 Clicker.cs 文件在 **MonoDeveloper** 编辑器中打开。
5. 现在编辑这个脚本文件：

```
using UnityEngine;
using System.Collections;

public class Clicker {
  public bool clicked() {
    return Input.anyKeyDown;
  }
}
```

6. 保存文件。

如果你是在为移动设备上的 Google Cardboard 开发，可以为 Cardboard 集成的触发器添加一个检查：

```
using UnityEngine;
using System.Collections;

public class Clicker {
  public bool clicked() {
#if (UNITY_ANDROID || UNITY_IPHONE)
    return Cardboard.SDK.CardboardTriggered;
#else
    return Input.anyKeyDown;
#endif
  }
}
```

我们写的任何脚本中如果需要用户点击的话都会用到这个 Clicker 文件。我们已经把用户点击的定义分离成一个单独的脚本文件，这样如果我们改变或重新定义用户点击的话就只需要改变这个文件。

3.6 虚拟现实设备的运行原理

戴上头盔，体验透视图吧！出现了 3D，感受 3D，可能你会有一种真正处于这个合成场景中的感觉。我猜想这不是你第一次体验虚拟现实，但是既然我们一起体验了，就让我们花几分钟讨论一下它的原理。

显而易见的是，虚拟现实看起来真的很酷！那它是怎么做到的呢？

*沉浸感*和*存在感*这两个词用来描述虚拟现实体验的特性。Holy Grail 就是用来增强这两个特性的，让它看起来非常真实，让你忘记了是在虚拟世界中。*沉浸感*是模拟你身体接收的（视觉的、听觉的、运行的等）感官输入感知的结果，这可以从技术上解释得通。而*存在感*是你被放在一个环境中的本能感——一种深刻的情感或直觉。你可以说沉浸感是 VR 的科学，而存在感是艺术且很酷。

很多不同的技术和技巧聚合在一起使虚拟现实体验运作起来，可以分成两个基本域：

1. 3D 视觉。

2. 头部动作跟踪。

换句话说，也就是显示器和传感器，就像现在手机中的显示屏和传感器，这就是为什么现在可以负担得起虚拟现实的一个很大的原因。

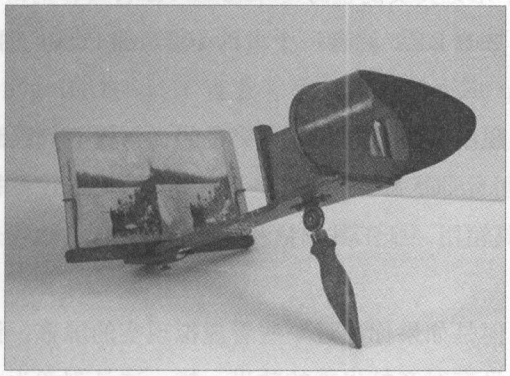
假设虚拟现实系统可以随时准确地知道你头部的位置，假设系统可以立即立体地渲染和显示这个精确视角的 3D 场景，那么无论何时何地你移动了，你都可以看见虚拟场景，都会有一个几乎完美的虚拟现实视觉体验。这就是它的基础，哈！

好了，我们慢一点来，逐字解释。

3.6.1 3D 立体视图

分屏立体画是在摄影术发明之后不久被发现的，就像 1876 年展出的图片（B. W. Kilborn & Co, Littleton, New Hampshire，参见 http://en.wikipedia.org/wiki/Benjamin_W._Kilburn）中的那个非常受欢迎的立体画查看器。一幅有立体感的照片为左眼和右眼分成两个视图，这两个视图有略微偏移以造成视差，这种欺骗让大脑以为它是真的 3D 视图。设

备也为双眼准备了两个镜头，让你很容易聚集在两张靠近的照片上。



类似地，渲染这些并排的立体图是 Unity 中开启了虚拟现实摄像机的第一件事。

假设你正戴着一个虚拟现实头盔并且静静地抱着你的头，那这张图片看起来就像是静止的，会比一张简单的立体图看起来要好一些。为什么？

那张过时的立体图中有两张相对小一点的带有矩形边框的孪生照片，当你的眼睛聚焦在视图的中心时，3D 效果令人信服。但是你会看到视图的边界，把你的眼球向四周移动（保持头部静止），剩余的沉浸感完全消失了。你现在只是一个在外边窥视透视图的观察者。

现在，不戴头盔的 Oculus Rift 的屏幕看起来是这样的（见下图）：

第一件你会注意到的事情是每只眼睛都对应一个桶形的视图。为什么是这样的呢？头盔的镜头是一个非常宽角度的镜头，所以，当你透过镜头看图的时候也会有一个非常宽的视野。其实，它是非常宽的（也非常高），它使图片变形（**枕形失真**，pincushion effect）。图形软件（SDK）对变形做了一次逆（inverse）操作（**桶形变形**，barrel distortion），这

样透过镜头看起来就正对我们了，这也被称作一次**视觉变形纠正**（ocular distortion correction），结果是一个未必真实的**视野**（Field of View，FOV），它足够宽到容纳很多边缘视觉。比如，Oculus Rift DK2 就有一个大约 100° 的 FOV。（我们将在第 9 章中深入讨论 FOV。）

另外当然了，两只眼睛的视角是稍有点错位的，与你两只眼睛间的距离或者说**瞳孔间距**（Inter Pupillary Distance，IPD）值相差无几。IPD 用于计算视差，每个人的值都不相同。（Oculus 配置工具箱用一个工具测量和配置你的 IPD，也可以让眼医测一个精确的距离。）

可能不那么明显，但是如果你离近一些看虚拟现实的屏幕，你会发现色彩分离，就像彩色打印机的打印头没有对齐打出来的效果一样，这是刻意为之的。光穿透镜头时会根据光的波长产生不同角度的折射。而渲染软件会对颜色分离再做一次逆操作，让它看起来正对我们，这也被称作为**色差校正**（chromatic aberration correction），这样可以使图片看起来**清晰**（really crisp）。

屏幕的分辨率也是得到一个有说服力的视图的重要因素。如果分辨率太低会看到像素，或者有些人称之为**纱窗效应**（screen door effect）的东西。显示器的像素宽高是对比头盔显示器时一个经常被引用的规格，但是**每英寸**[⊖]**像素数**（ppi）值可能更重要。显示器技术的其他创新点比如**像素拖尾**（pixel smearing）和**注视点渲染**（foveated rendering）（恰好当你眼球观看时显示高分辨率的细节）也能够帮助减少纱窗效应。

当你用虚拟现实技术体验 3D 场景时，你还必须考虑**每秒帧数**（Frames Per Second，FPS）。如果 FPS 太低，动画就不是很连贯。影响 FPS 的因素包括图形处理器（GPU）的性能和 Unity 的场景的复杂度（多边形和光照计算的数量），还有一些其他因素。这在虚拟现实中是较复杂的，因为你需要绘制两次场景，为每只眼各绘制一次。技术上的创新，比如对虚拟现实的 GPU 优化、帧插值（frame interpolation）及其他技术，可以改进帧率。对于我们开发者来说，Unity 中的移动游戏开发者用到的那些性能调优技术也可以用于虚拟现实。（我们将在第 8 章中深入讨论性能优化。）这些技巧和光学让 3D 场景显得逼真。

声音也非常重要，比很多人所认识的更重要。我们应该戴着立体声耳机体验虚拟现实。其实，当音频很棒而图像很糟糕时，你仍然可以有一个很好的体验，这种现象在电

⊖ 1 英寸 = 2.54 厘米。——译者注

视和电影院中很常见，在虚拟现实中同样适用。双耳音频（binaural audio）以这样方式给双耳提供其声源的立体视图，让你的大脑想象想象它在 3D 空间中的位置。不需要特别的收听设备，一般的耳机就可以（扬声器不可以）。比如，戴上你的耳机在这个网址 https://www.youtube.com/watch?v=IUDTlvagjJ 查看《Virtual Barber Shop》。真 3D 音效，比如 **VisiSonics**（由 Oculus 颁布许可），提供一种更逼真的空间音频渲染。声音在临近的墙上反弹后会被场景中的障碍物阻挡，以增强第一人称的体验和现实感。

最后，虚拟现实头盔的大小应该刚好适合你的头和脸，让你感觉不到正戴着它，并且应该遮挡你周围真实环境中的光。

3.6.2　头部跟踪

那么，假设我们有一张好看的 3D 图片，可以在一个舒适的带有广角的虚拟现实头盔中查看，你移动头部就会感觉到有一个透视图盒子粘在你的脸上。移动头部时盒子也随之移动，这就像你手里拿着古董级的立体画设备或者小时候玩的**魔景机**（View Master）。幸运的是，虚拟现实要酷得多。

虚拟现实头盔内部有一个动作感应器（IMU），用于检测空间加速度和三个轴上的转速，叫作**六自由度**（six degrees of freedom）。这也是在手机和一些控制台游戏控制器里同样常用的技术，安装在头盔上，当你移动头部时，会计算当前视角（viewpoint），下一帧画面绘制时使用此视点，这也被称作**运动检测**（motion detection）。

如果你想在手机上玩游戏，那么当前的动作感应器也许还行，但是对于虚拟现实，它还不够精准，误差（化整误差）会随着时间而累积，因为感应器每秒钟要获取几千次样本，偶尔会丢失对你在真实世界中的跟踪。*漂移*（drift）问题是基于手机的虚拟现实头盔的主要偏差，比如 Google Cardboard，它能够感应你头部的动作，但是它会丢失对你头部位置的跟踪。

高端的头盔显示器用分离的位置跟踪（positional tracking）机制解决飘移问题。Oculus Rift 用一个由内向外的定位跟踪器（inside-out positional tracking）解决这个问题，在头盔显示器上有一组（不可见的）红外 LED，通过外部的光感应器（红外摄像头）读取这些 LED 的值，从而决定你的位置。你需要保持在摄像机的视图中让头部跟踪能够运行。

另外，Steam VR Vive Lighthouse 技术公司制作了一个由外向内的定位跟踪器，有两个或多个不发声的激光发射器放在房间中（更像是食品杂货店收银台的二维码扫描器

中的激光），然后有一个光感应器在头盔上读取这些激光的射线以确定你的位置。

两种方法的主要目的都是精确地定位你头部的位置（以及其他类似配备的设备，比如手持控制器）。

位置、倾斜和头部的正方向，或者说头部姿势（head pose），被图形软件用于在这个便于观察的位置上重绘 3D 场景。诸如 Unity 这样的图形引擎擅长做这个。

现在，假设屏幕正以每秒 90 帧的速度刷新，并且你正在移动你的头部，软件确定头部姿势、渲染 3D 视图并绘制在头盔显示器的屏幕上，而你还在不断地移动头部，所以在显示的时候，图片相对于你当前的位置有点过时，这叫作**延迟**（latency），它让你感觉到恶心呕吐。

晕动症（cmotion sickness），是当你在虚拟现实中移动头部时产生延迟，但你的大脑却期望你周围的事物随之同步变化而导致的。至少可以这么说，任何可感知的延迟都会让你感到不舒服。

延迟可以用从读取动作传感器的值开始到渲染出相应图片的时间来衡量，或者说是传感器到像素（sensor-to-pixel）的延迟，根据 Oculus 的 John Carmack 所说：

> “总量为 50 毫秒的延迟是可响应的，但还是有明显的延迟感。20 毫秒或更短的延迟水平被认为是可接受的。”

有很多非常聪明的方法可用于实现延迟抵消（letancy compensation），具体的细节超出了本书的范围，且必然会随着设备厂商的技术改进而变化。其中之一是 Oculus 称为时间**扭曲**（timewarp）的方法，它尝试预测渲染完成时你的头部的位置，并且使用那个预测的头部姿势取代真实检测到的头部姿势。所有这些都由 SDK 处理，所以作为 Unity 开发者，不需要直接处理它。

同时，作为虚拟现实开发者，我们也需要知道像延迟这样导致晕动症的原因，延迟可以通过快速渲染帧（保持建议的 FPS）来减少。而快速渲染帧可以通过抑制头部移动太快和其他让用户感觉到接地和舒适的技术实现。晕动症（motion sickness）在第 6 章中会深入讨论。

Rift 所做的改进头部跟踪和现实感的另一件事是使用一个颈部的骨骼表示（skeletal representation），让它接受的所有旋转更精确地映射到头部的旋转。比如，向下看你的膝盖时会产生一点向前的位移，因为它知道你几乎不可能立刻向下旋转头部。

除了头部跟踪、立体画和 3D 音效之外，虚拟现实的体验还可以用身体跟踪、手部跟踪（以及手势识别）、运动跟踪（如虚拟现实跑步机）和带触觉反馈的控制器而加强。

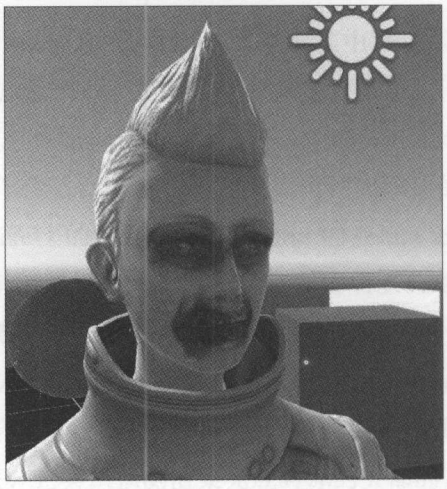
所有这些技术的目标都是增加你的沉浸感和虚拟世界中的现实感。

小结

本章中，我们讨论了不同级别的设备集成软件，然后把适合你的目标虚拟现实设备的集成软件安装上。安装过程不一定会很顺利，但是已经有很多先驱者如此操作过，所以如果遇到障碍了可以在互联网上得到帮助。我们还讨论了硬件的内部运作机制，以及让虚拟现实运作并且对我们虚拟现实开发者很有用的软件 SDK。

Chapter 4

第 4 章
基于凝视的操控



到目前为止，我们的透视图是一个第三人称的虚拟现实体验。当你进入场景中，你就像是一个观察者或者第三人称摄像机。当然，你可以四处查看，并且通过增加控制来移动摄像机的视角。然而，任何在场景中的动作都是从第三人称角度发出的。

本章，我们还将保持第三人称的模式，但是会多一些亲自参与。我们会探索在你的虚拟世界中通过扫视和凝视控制物体的技术。我们的主角——Ethan，将会受你控制，对你看的地方做出回应。此外，我们还将开始编写 Unity 脚本程序。在这个过程中，我们将讨论以下话题：

- 添加 **AI** 和 **NavMesh** 到我们的第三人称角色 Ethan 身上。
- 使用 C# 进行 Unity 编程。
- 使用我们的眼睛来移动 3D 光标。
- 完美实现射杀僵尸 Ethan。

4.1　步行者 Ethan

很多 VR 的切入点都是游戏。所以，我们也不能免俗。我们将赋予角色 Ethan 一条命。嗯，近似（或者不是）算是一条命吧，因为他会变成一个僵尸。

我们将离开 Ehtan 出现的那个透视图。如果你有键盘或者手柄，你可以让他围着场景跑，但是目前的 VR 还不能保证。实际上，如果你使用 Google Carboard 观察一个场景，你不太可能有一个手持控制器（即使是蓝牙游戏控制器）。我们还是正视现实。即使使用 Oculus Rift，也必须使用键盘或者一个手柄控制器且都是非常不友好的，因为你看不到自己的双手。也许，可以有另一种方式让他移动。有一种技术就是当你戴着 VR 头盔的时候，使用你的凝视方向。

在我们尝试这种方法之前，我们会先把 Ethan 变成僵尸让他不受控制地漫无目的四处游走。我们通过给他一些 AI 并写一个脚本发送给他来实现让他随机地游走。

4.1.1　人工智能 Ethan

开始之前，我们想用 Unity 的 AI 角色 AIThirdPersonController 替换之前的 ThirdPersonController 预制件，使用下面的步骤。Unity 粗略地使用词语人工智能 artificial intelligence 表示脚本驱动。执行下面的步骤：

1. 打开上一章的 Diorama 场景，然后从 **Standard Assets** 中导入 **Character** 包。

2. 在 **Project** 面板中，打开 Standard Assets/Characters/ThirdPersonCharacter/Prefabs 文件夹，并且将 AIThirdPersonController 拖入场景，命名为 Ethan。

3. 在 **Hierarchy** 面板中（或者 **Scene** 中），选中上一个 ThirdPersonController（那个老的 Ethan）。然后，在 **Inspector** 面板的 **Trasform** 中，选择右上角的齿轮图标，并且选择 **Copy Component**。

4. 选中新 Ethan 对象（从 **Hierarchy** 或者 **Scene** 中）。然后，在 **Inspector** 面板的 **Transform** 中，选择齿轮图标，并且选择 **Paste Component Values**。

5. 现在你可以通过在 **Hierarchy** 面板中选中老的 Ethan，右键点击打开选项卡，点击 **Delete** 来删除它。

注意，这个控制器拥有一个 **NavMesh Agent** 组件和一个 AICharacterControl 脚本。NavMesh Agent 有一些表示 Ethan 如何围绕场景移动的参数。AICharacterControl 脚本持有 Ethan 要到达的目标。让我们把这些组装起来：

1. 添加一个空的游戏对象到 **Hierarchy** 面板。

2. 重置它的 **Transform** 值。

3. 命名为 WalkTarget。

4. 选中 Ethan 并且拖动 WalkTarget 到 **Inspector** 面板的 **AI Character Control** 项的 **Target** 属性中。

进行到这一步，我们的场景中就有了一个 AI 角色（Ethan），一个空的游戏对象，作为导航目标（WalkTarget）。同时，我们也告诉了 AI 角色控制器去使用这个目标对象。当我们运行游戏的时候，无论 WalkTarget 在哪里，Ethan 都会走到那里。但是现在还不行。

4.1.2 Navmesh 烘焙

在告诉 Ethan 哪些地点是允许游走的之前，它是不能四处走动的。我们需要定义一个“NavMesh“——一个简化的几何平面，它允许一个角色环绕障碍物绘制出路径。

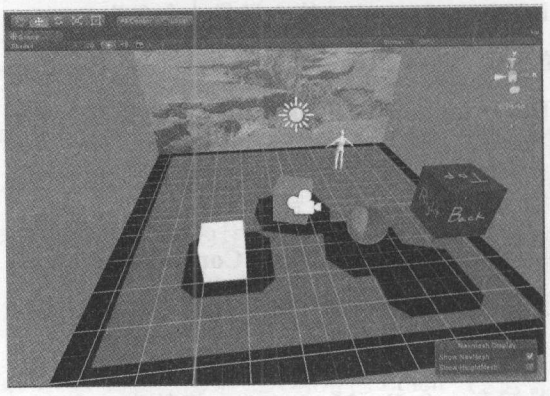
先通过识别场景中的静态物体来创建 NavMesh，然后烘焙它：

1. 选中所有的物体——地面、三个立方体和球体，然后在 **Inspector** 面板中反选 **Static** 复选框（你可以对每个对象单独操作或者使用 Ctrl+ 点击实现一次选中多个）。

2. 选中 **Navigation** 面板。如果它还不在你的编辑器中，从主菜单中选择 **Window | Navigation** 打开“Navigation”窗口。

3. 点击面板底部的 **Bake** 按钮。

场景的视图现在看上去被蓝色覆盖，这个就是定义的 NavMesh，如下图所示：



让我们测试一下。保证 **Game** 面板的 **Maximize on Play** 未被选中。点击 Play 按钮（编辑器顶部的三角形）。在 **Hierarchy** 面板中，选中 WalkTarget 对象然后确保 Translate 小部件在 **Scene** 面板中是激活的（按下 W 键）。现在拖动红色（X）及（或者）蓝色（Z）

箭头操作 WalkTarget 对象用于在地板周围而移动。Ethan 将会随着你而移动！再次点击 Play 暂停试玩模式。

4.1.3　镇上的游走者

现在，我们将写一个脚本来移动 WalkTarget 到一个随机的地方。

如果你以前鼓捣过 Unity，你肯定写过一些脚本。我们将使用 C# 作为编程语言。我们会慢慢来写第一个脚本。我们会将 WalkTarget 与脚本进行连接，具体步骤如下：

1. 在 **Hierarchy** 面板或者 **Scene** 视图中选择 WalkTarget 对象。
2. 点击 **Inspector** 面板中的 **Add Component** 按钮。
3. 选择 **New Script**。
4. 命名为 RandomPosition。
5. 选择 C# 作为编程语言。
6. 点击 **Create** 和 **Add**。
7. 这将在 WalkTarget 对象上创建一个脚本。双击 **Inspector** 面板中 **Script** 右边的槽中的 RandomPosition 脚本，在 Script 的右边，在 **MonoDevelop** 代码编辑器中打开。

4.1.4　插曲——Unity 编程简介

Unity 能做很多事情——管理对象、渲染对象、给对象添加动画、计算对象的物理，等等。Unity 本身是个程序，它也是由代码创造的——可能是一群非常聪明的人写的非常棒的代码。你一游戏开发者，可以访问这些内部的 Unity 代码，就像我们之前操作的通过可点击的 Unity 编辑器界面使用它。在 Unity 编辑器中，脚本被表现为可配置的组件。然而，这也让你可以通过 Unity 的脚本 API 进行更多直接的访问。

API（Application Programming Interface）指的是已发布的可以在脚本中使用的软件函数。Unity 的 API 非常丰富并且设计优良。这也是为什么人们为 Unity 写了很多插件。

市面上有很多种编程语言。Unity 选择支持来自微软的 C# 语言（还有 JavaScript，但其功能有限）。计算机语言有必须遵循的特殊语法。否则计算机将不能读懂你的脚本。在 Unity 中，脚本错误（和警告）将会显示在 **Console** 面板中，即在编辑器的底部。

默认的脚本编辑器是一个集成开发环境（IDE），称为 **MonoDevelop**。你可以配置其

他编辑器或者 IDE，如果你想的话，像微软的 Visual Studio 也行。MonoDevelop 有一些很棒的功能，如自动补全以及弹出提示帮助你理解 Unity 文档。C# 脚本是一些文本文件，以 .cs 后缀命名。

在 Unity C# 脚本中，有一些单词和符号是 C# 语言自身的一部分，一些来自微软 .NET 框架，另一些是由 Unity API 提供的，再就是你要写的代码。

一个默认的 Unity C# 脚本是如下这样的：

```
using UnityEngine;
using System.Collections;

public class RandomPosition : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

让我们来剖析一下。

前两行指示这个脚本需要一些其他东西才能运行。Using 关键字属于 C# 语言。Using UnityEngine 这一行，说的是我们将使用 UnityEngine API。Using System.Collection；意思是我们可能会使用名称为 Collections 的类库用于访问集合对象。

在 C# 中，每一行代码的都以分号结尾。双斜杠表示代码中的注释，任何从它开始到行末的内容都会被忽略。

这个 Unity 脚本定义了一个名为 RandomPosition 的类。**Classes** 类似于代码模板，拥有自己的属性（变量）和行为（方法）。该类继承 MonoBehaviour 基类，可以被 Unity 识别，并且在你的游戏运行中被使用。Public class RandomPosition: MonoBehaviour 指的是，我们正在定义一个新的公有类，名为“RandomPosition”，它集成了“MonoBehaviour”这个 Unity 基类的所有能力，包括 Start() 和 Update() 方法。类的内容被一对封闭的大括号包含。

当声明为 public 的时候，它可以被这个指定的脚本文件之外的代码访问。当它是

private 的时候，它只能被这个文件引用。我们想让 Unity 能够访问 RandomPosition 类。

类定义变量及函数。一个 **variable** 保存一种特定类型的数值，比如 float、int、boolean、GameObject、Vector3 等。**Functions** 实现逻辑（一步步的指令）。函数可以接收参数（包含在小括号中被函数中的代码使用），还可以在函数结束时返回一个新的值。

数值型的 float 常量，例如 5.0f，在 C# 中需要在结尾写一个 f 来保证数据类型是一个单精度的浮点型数据，而不是一个双精度浮点类型。

Unity 会自动调用某些特殊的消息方法，前提是你已经定义过。Start() 和 Upldate() 方法就是两个例子。在默认的 C# 脚本中提供了空方法体。一个方法之前的数据类型表示这个方法将要返回的数据类型。Start() 和 Update() 都不返回数据，所以它们的类型是 void。

在你的游戏开始之前，所有的 MonoBehaviour 脚本中的 Start() 函数都会被调用。这是做数据初始化的好地方。游戏运行过程中，在每个时间片或者帧之间 Update() 方法都会被调用。大多数操作会放在这里。

一旦你在 MonoDevelop 中写下或者修改了一个脚本，请保存它。然后，转到 Unity 编辑器窗口。Unity 将会自动识别到脚本文件被修改，并且重新导入。如果发现了错误，它将会在 **Console** 面板中立即报告出来。

这就是一些简单的 Unity 编程。随着本书内容的深入，我会在涉及的时候进行详细解释。

4.1.5　RandomPosition 脚本

现在我们想移动 WalkTarget 对象到一个随机地点，让 Ethan 面向那个方向，几秒钟之后，再移动到 WalkTarget 对象一次。这样，他就会看起来是漫无目的地徘徊。我们可以使用一个脚本来实现，下面将逐行解释。RandomPosition.cs 脚本如下：

```
using UnityEngine;
using System.Collections;

public class RandomPosition : MonoBehaviour {

  void Start () {
    StartCoroutine (RePositionWithDelay());
  }

  IEnumerator RePositionWithDelay() {
    while (true) {
```

```
        SetRandomPosition();
        yield return new WaitForSeconds (5);
    }
}

void SetRandomPosition() {
    float x = Random.Range (-5.0f, 5.0f);
    float z = Random.Range (-5.0f, 5.0f);
    Debug.Log ("X,Z: " + x.ToString("F2") + ", " +
     z.ToString("F2"));
    transform.position = new Vector3 (x, 0.0f, z);
}
}
```

该脚本定义了一个名为 RandomPosition 的 MonoBehaviour 子类。我们定义一个类的第一件事是声明一些将要使用的变量。一个变量是一个值的占位符。数值可以在这里初始化或者在其他地方赋值，只要保证脚本在使用它的时候它拥有一个值就行。

这个脚本最实用的地方是下面名为 SetRandomPosition() 的方法。让我们看看它做了什么。

回想一下，GroundPlane 平面是 10×10 的单位矩形，原点位于中间。所以，任何在平面上的 (X, Z) 点都会各个轴上的 [-5, 5] 区间。代码行 float x= Random.Range(-5.0f, 5.0f); 在给定的区间内取一个随机值并且赋值给一个新的 float x 变量。用同样的办法我们可以获得随机的 z 值。(通常，我不鼓励写死一个常量值，而是像现在这样使用变量，但是为了解释原理我会尽量保持简单。)

代码行 Debug.Log("X,Z:"+x.ToString("F2")+","+z.ToString("F2"); 会在游戏运行时将 x 和 z 的值打印到 **Console** 面板中。你会看到类似于 X, Z:2.33, -4.02 这样的内容，因为 ToString("F2") 表示小数点后保留两位。注意我们使用的是 + 表示将输出字符连接在一起。

代码行 transform.position=new Vector3(x,0.0f,z); 表示我们真正要将目标移动到给定的地点。我们设置这个脚本连接到的对象的变换位置。在 Unity 中，有 X, Y, Z 的值都用 Vector3 对象表示。所以，我们创建了一个有 X 和 Z 值的对象。$Y = 0$ 表示它躺在 GroupdPlane 上。

每一个 MonoBehaviour 类都有一个内置的变量 this，它指向脚本连接的那个对象。也就是说，当一个脚本是对象的一个组件并且出现在 **Inspector** 面板中的时候，这个脚本可以用 this 指向它的对象。实际上，this 是很明显的，如果你想调用 this 对象

上的方法，你不需要去声明。我们应该已经看到了 `this.transform.position=...`，但是这个 this 对象是隐式的，并且通常是隐藏的。另一方面，如果你有一个变量可以用于其他对象（例如，GameObject that），那么你最好在设置它位置的时候这样写：`that.transform.position =...`。

最后一个难点是我们如何在 Unity 中处理时间延迟。在我们这个例子中，变换的位置应该每 5s 变化一次。可以通过这几步来解决：

1. 在 `Start()` 方法中，`StartCorountine(RepositionWithDelay());` **coroutine** 是一小段在被调用时与方法分开运行的代码，它来自于一个调用的方法。所以，这行代码会启动 `RepositionWithDelay()` 方法，把它放入协程中。

2. 在内部，有一个 `while(true)` 循环，你可能猜到了，它会一直运行（只要游戏在运行中）。

3. 调用 `SetRandomPosition()` 方法，重新定位对象。

4. 然后，在这个循环的底部，我们使用 `yield return new WaitForSecond(5);` 语句，它会告诉 Unity，嘿，你有 5s 的时间做你想做的事情，然后回到这里，我开始下次循环。

5. 为了让这些都能运行，`RePositionWithDelay` 协程必须声明为 `IEnumerator` 类型（因为文档中要求这样）。

这个 co-routine/yield 机制虽然是高级编程的话题，但在时间片程序（比如 Unity）中是一个通用的模式。

我们的脚本将会保存在名为 `RandomPosition.cs` 的文件中。

现在可以继续了。在 Unity 编辑器中，点击 Play。Ehtan 会像一个疯子一样从一个地方跑到另一个地方。

4.1.6　“僵尸” Ethan

好了，真是相当随机。让我们调整一下 NavMesh 的操作参数让他慢下来做一个类似于僵尸的步速。我们要做以下的步骤：

1. 在 **Hierarchy** 面板中选中 Ethan。

2. 在 **Inspector|Nav Mesh Agent | Steering** 中，设置：

- **Speed**：0.3
- **Angular Speed**：60

❑ **Acceleration：2**

再次点击试玩。他慢下来了。不错。

另外一个点睛之笔——把他变为僵尸。可以用一张名为 EhtanZombie.png 的纹理图片（包含在本书中）。执行以下步骤：

1. 在主菜单中的 **Assets** 中选择 **Import New Asset...** 定位到本书中的资源文件夹中。

2. 选中 EthanZombie.png 文件。

3. 点击 **Import**。为保持整洁，保证它位于 Assets/Texture 文件夹中。（你也可以将文件从 Windows Explorer 拖入 **Project** 面板的文件夹。）

4. 在 **Hierarchy** 面板中，展开 Ethan 对象（点击三角形），并选择 EthanBody。

5. 在 **Inspector** 面板中，点击 **Shader** 左边的三角形图标，展开 EthanWhite 着色器。

6. 在 Project Assets/Texture 文件夹中选择 EthanZombie 纹理。

7. 将它拖动到 **Albedo** 纹理贴图上。它是 **Main Maps** 下的 **Albedo** 标签左边的一个小正方形。

8. 在 **Hierarchy** 面板中选择 EthanGlasses，再在 **Inspector** 面板中反选它以禁用眼镜。毕竟，僵尸是不需要眼镜的！

他的形象在本章开头已经展示过了！你说什么？那是个跛脚的僵尸？好吧，他可能是刚刚变化的。从头再来你可以自己制作一个更好的。使用 Blender、Gimp 或 PhotoShop 并且自己上色（甚至你可以导入一个完全不同的僵尸模型替换 EthanBody）。

现在，构建项目并且在 VR 中试试。

我们使用第三人称视角观看。你可以观看四周并且能看到正在发生的事情。这很有趣，也很好玩。他是被动的，让我们把他变得更主动些。

4.2 向我看的方向行走

在下一个脚本中，我们将把 Ethan 送到我们正在看的地方，而不是随机走动。在 Unity 中，使用**光线投射**（ray casting）已经很完善了，类似于从摄像机中射出一束光并且能够看到它击中了什么（更多信息，请访问 http://docs.unity3d.com/Manual/CameraRays.html）。

我们将创建一个新的脚本，就像之前一样我们把它附加到 WalkTarget 上：

1. 在 **Hierarchy** 面板中或者 **Secne** 视图中选中 WalkTarget 对象。

2. 在 **Inspector** 面板中，点击 **Add Component** 按钮。

3. 选择 **New Script**。

4. 命名为 LookMoveTo。

5. 确保选择 **C# 语言**。

6. 点击 **Create** 和 **Add**。

这将会在 WalkTarget 对象上创建一个脚本组件。双击使其在 MonoDevelop 代码编辑器中打开。

4.2.1　LookMoveTo 脚本

在我们的脚本中，每当 `Update()` 方法被调用时，我们将会读取摄像机面向的位置（使用变换位置值和旋转值）。在这个方向上投射一束光线，然后让 Unity 告诉我们它打在了地平面的哪个位置点上。然后，我们使用这个位置点来设置 WalkTarget 对象的位置。

这是完整的 LookMoveTo.cs 的脚本：

```
using UnityEngine;
using System.Collections;

public class LookMoveTo : MonoBehaviour {
  public GameObject ground;

  void Update () {
    Transform camera = Camera.main.transform;
    Ray ray;
    RaycastHit hit;
    GameObject hitObject;

    Debug.DrawRay (camera.position,
      camera.rotation * Vector3.forward * 100.0f);

    ray = new Ray (camera.position,
      camera.rotation * Vector3.forward);
    if (Physics.Raycast (ray, out hit)) {
      hitObject = hit.collider.gameObject;
      if (hitObject == ground) {
        Debug.Log ("Hit (x,y,z): " + hit.point.ToString("F2"));
        transform.position = hit.point;
      }
    }
  }

}
```

让我们逐步地看一下这个脚本：

```
public GameObject ground;
```

脚本首先为 GroundPlane 定义了一个变量。因为它为 public，所以我们可以使用 Unity 编辑器将它赋值为实际的对象：

```
void Update () {
  Transform camera = Camera.main.transform;
  Ray ray;
  RaycastHit hit;
  GameObject hitObject;
```

在 Update() 方法中，我们定义了一些局部变量：camera、ray、hit 和 hitObject，这些变量的数据类型是我们将要用到的 Unity 方法中所需要的。

Camera.main 是现在正在使用的摄像机对象（也就是标记为“MainCamera”的那个）。我们获取它当前的 transform 值，它将会被赋值给摄像机变量：

```
ray = new Ray (camera.position,
  camera.rotation * Vector3.forward);
```

先暂时忽略方便的 Debug 语句，我们首先使用 new Ray() 主射线从 camera 发出。

ray 可以由 *X*，*Y*，*Z* 空间上的起点和一个方向向量来定义。**direction vector** 可以定义成空间中的一个 3D 起点到另一个终点的相对位移。向前的方向，也即 *Z* 轴的正方向（0, 0, 1）。Unity 会帮我们计算。所以，如果我们使用一个单位向量（Vertor3.forwad），乘以 3 轴旋转（camera.rotation），并且以长度（100.0f）进行缩放，我们将获得一个与摄像机相同方向射出的 100 个单位长度的射线。

```
if (Physics.Raycast (ray, out hit)) {
```

然后，我们将这束光线投射出去，然后看看它是不是遇到什么东西。如果遇到，hit 变量应该会保存着它遇到的东西的更多细节，其中包括那个对象 hit.collider.gameObject。（out 关键字表示 hit 变量的值会被 Phisics.Raycast() 方法赋值。）

```
if (hitObject == ground) {
  transform.position = hit.point;
}
```

我们检查光线是否击遇到了 GroundPlane 对象，如果遇到，我们将会把 WalkTarget 对象移动到相遇的那个 hit 的点上。

“==”比较操作符与“=”运算符不同，不要混淆，“=”是赋值运算符。

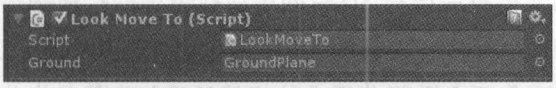
脚本包含了两个 Debug 语句，它们用于在 Play 模式下监听脚本做了什么。Debug.DrawRay( ) 将会在 **Scene** 视图中绘制出给定的射线，这样你可以真实地看到它，Debug.Log( ) 将会将现在的交点发送到控制台（如果有交点的话）。

保存脚本，切换到 Unity 编辑器，执行以下步骤：

1. 选中 WalkTarget，在 **Inspector** 面板中，LookMoveTo 脚本组件现在有一个字段用于 GroundPlane 对象。

2. 在 **Hierarchy** 面板中，选择并且拖动 GroundPlan 游戏对象到 **Ground** 字段中。

保存场景，脚本面板看起来如下图。



然后，点击 Play 按钮。Ethan 将会跟随我们的视线（以他自己的步频）。

4.2.2　添加反馈光标

你的视线击中平面上的哪个点并不总是很明显，我们现在要添加一个光标到场景中。做法非常简单，因为我们刚刚做的是在一个不可见的空的 WalkTarget 周围移动。如果我们用下面的步骤给它一个网格，它将会可见：

1. 在 **Hierarchy** 面板中，选择 WalkTarget 对象。

2. 点击鼠标右键，选择 **3D Object|Cylinder**。这将会创建一个圆柱体，其父对象为 WalkTarget。（同样，你可以使用主菜单上的 **GameObject** 选项卡，然后拖动对象到 WalkTarget 上面。）

3. 点击 **Transform** 面板上的齿轮图标，点击 **Reset**，确保我们以默认的变换值开始。

4. 在 **Inspector** 面板中选中圆柱体，将 **Scale** 改为（0.4, 0.05, 0.4）。这将会创建一个直径为 0.4 的平面圆盘。

5. 反选 **Capsule Collider** 复选框，使之不可用。

6. 在 **Mesh Render** 中，你同样可以禁用 **Cast Shadows**、**Receive Shadows**、**Use Light Probes** 和 **Reflection Probes**。

现在，再次试玩。指示光标盘将会跟随我们的视线。

如果你愿意，可以使用一个有颜色的材质来装饰圆盘。更好的方案是，选择一个合

适的纹理。比如我们在第 2 章中为 GridProjector 文件（Standard Assets/Effects/Projectors/Textures/Grid.psd）使用的网格纹理。本书提供了一个 CircleCrossHair.png 文件。拖动纹理到圆柱体光标上。操作的时候，把 **Shader** 设置为 **Standard**。

4.2.3 观察者

在这个项目中，我们通过移动 WalkTarget 对象到地平面上的一个位置，这个位置由摄像机发出的射线与地平面相交获得，使 Ethan 跟随我们的视线。

你会发现当我们的视线扫过立方体和球体的时候，光标会被卡住。那是因为**物理引擎**（physics engine）检测到了哪个物体首先被击中，射线将永远到不了地平线。在我们的脚本中，在移动 WalkTarget 之前设置了条件语句 `if(hitObject==ground)`。如果不这么做，我们的光标将会漂在 3D 空间中射线击中的任何物体之上。有时候这样很有意思，但是在我们的例子中并不有趣。我们希望光标在地平面上。不过现在如果光线没有击中地平面上，它将不会重新获得点位，并且看起来像是卡住了。你可以想个办法解决它吗？这里有个提示：翻翻 `Physics.RaycastAll`。好了，我来演示给你看。将 Update( ) 中的代码替换成下面的代码：

```
Transform camera = Camera.main.transform;
Ray ray;
RaycastHit[] hits;
GameObject hitObject;

Debug.DrawRay (camera.position, camera.rotation *
  Vector3.forward * 100.0f);

ray = new Ray (camera.position, camera.rotation *
  Vector3.forward);
hits = Physics.RaycastAll (ray);
for (int i = 0; i < hits.Length; i++) {
  RaycastHit hit = hits [i];
  hitObject = hit.collider.gameObject;
  if (hitObject == ground) {
    Debug.Log ("Hit (x,y,z): " +
      hit.point.ToString("F2"));
    transform.position = hit.point;
  }
}
```

在调用 `RaycastAll` 时，我们会得到一个列表或一个数组，一个击中的列表。然

后，我们循环查找每一个沿着光线路径上的地平面上的交点。现在，我们的光标将能够沿着地平面追踪，无论这中间有没有其他物体。

额外挑战：另外一个更有效的解决方案是使用 layer system。创建一个新的层，将它赋值给平面，然后将它作为一个参数传递给 `Physics.raycast()` 方法。你知道为什么这个方法有效得多吗？

4.3　如果眼神可以杀人

让我们继续。我们可能想试着击杀 Ethan（哈哈！）。下面是这个新功能的详细说明：

- 注视 Ethan 并使用我们的视线之枪击杀他。
- 当枪击中目标的时候发出火焰。
- 击中 3s 之后，Ethan 被杀死。
- 被杀死后 Ethan 将会爆炸（我们得到一个点），然后在一个新位置复活。

4.3.1　KillTarget 脚本

现在，我们将脚本附加到一个新的 GameController 对象中，如下：

1. 创建一个新的游戏对象并且命名为 GameController。
2. 使用 **Add Component** 附加一个新的 C# 脚本，命名为 KillTarget。
3. 在 MonoDevelop 中打开脚本。

这是完整的 KillTarget.cs：

```
using UnityEngine;
using System.Collections;

public class KillTarget : MonoBehaviour {
  public GameObject target;
  public ParticleSystem hitEffect;
  public GameObject killEffect;
  public float timeToSelect = 3.0f;
  public int score;

  private float countDown;

  void Start () {
    score = 0;
```

```
        countDown = timeToSelect;
        hitEffect.enableEmission = false;
    }

    void Update () {
        Transform camera = Camera.main.transform;
        Ray ray = new Ray (camera.position, camera.rotation *
            Vector3.forward);
        RaycastHit hit;
        if (Physics.Raycast (ray, out hit) && (hit.collider.gameObject
            == target)) {
            if (countDown > 0.0f) {
                // on target
                countDown -= Time.deltaTime;
                // print (countDown);
                hitEffect.transform.position = hit.point;
                hitEffect.enableEmission = true;
            } else {
                // killed
                Instantiate( killEffect, target.transform.position,
                    target.transform.rotation );
                score += 1;
                countDown = timeToSelect;
                SetRandomPosition();
            }
        } else {
            // reset
            countDown = timeToSelect;
            hitEffect.enableEmission = false;
        }
    }

    void SetRandomPosition() {
        float x = Random.Range (-5.0f, 5.0f);
        float z = Random.Range (-5.0f, 5.0f);
        target.transform.position = new Vector3 (x, 0.0f, z);
    }
}
```

让我们逐步解释。首先，我们声明了一些公有变量，如下：

```
public GameObject target;
public ParticleSystem hitEffect;
public GameObject killEffect;
public float timeToSelect = 3.0f;
public int score;
```

像我们上个 LookMoveTo 脚本中做的一样，我们的目标是 Ethan。我们还添加了一个 hitEffect 的粒子发射器、一个 killEffect 爆炸体和一个计算器的初始值 timeToSelect。我们将我们的击杀次数保存在 score 变量中。

Start() 在游戏开始时被调用，将 score 初始化为 0，设置 countDown 计时器为它的初始值，并且关闭 hitEffect。

然后，在 Update() 方法中，与 LookMoveTo 脚本一样，我们从摄像机中投射一束射线并且判断他是否击中了目标 Ethan。当这些都完成后，我们检查 countDown 计时器。

如果计时器还在计数，在 Update() 最后一次被调用的时候我们使用 Time.deltaTime 减去总的走过的时间值，同时保证 pickEffect 是从击中点发射出。

如果射线还在它的目标上并且计时器已经完成了倒计时，那么 Ethan 就被杀掉了，并且会爆炸，我们将 score 加 1，重新设置定时器为起始值，然后移动（复活）Ethan 到一个随机的新位置。

我们使用 Unity 中 ParrticleSystems 包中的标准资源来表现爆炸。要激活它，killEffect 应该被设置为预制件，并且命名为 Explosion。然后脚本将其实例化。换句话说，它使自己成为了场景中（以一个特定的变换值）的一个对象，并开启了完美的脚本和效果。

最后，如果射线没有击中 Ethan，我们将重置计数器并关闭粒子。保存脚本并进入 Unity 编辑器。

额外挑战：依照我们在本章开始的 RandomPosition 脚本中做的那样，使用协程重构脚本来管理延迟时间。

4.3.2　添加粒子效果

现在为了填充几个 public 变量，我们要执行以下步骤：

1. 首先，我们需要 Unity 标准资源中的 ParticleSystem 包。如果还没有添加，在 **Assets | Import Package | ParticleSystems** 中，选择 **All**，然后点击 **Import**。

2. 在 **Hierarchy** 面板中，选择 GameController，然后进入 **Inspector** 面板中的 **Kill target**（脚本）面板。

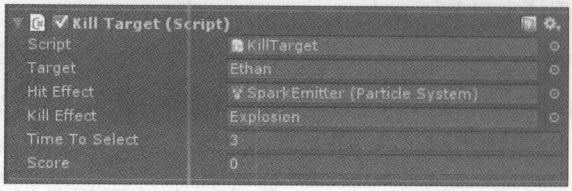
3. 从 **Hierarchy** 面板中，拖动 Ethan 对象到 **Target** 字段中。

4. 在主菜单中，找到 **GameObject | Particle System**，并且命名为 SparkEmitter。

5. 再次选中 GameController，然后将 SparkEmitter 拖动到 **Hit Effect** 字段中。

6. 在 **Project** 面板中，找到位于 Assets/Standard Assets/ParticlesSystems/Prefabs 中的 Explosion 预制件，拖动 Explosion 预制件到 **Kill Effect** 字段中。

脚本面板的截图如下：



我们创建了一个默认的粒子系统作为火光发射器，可以根据喜好来设置它。我会带你开始，同时你可以根据你的意愿来对其进行配置：

1. 选择 **Hierarchy** 面板中的 SparkEmitter。

2. 在它的 **Inspector** 面板中，**Particle System** 下面，设置以下值：

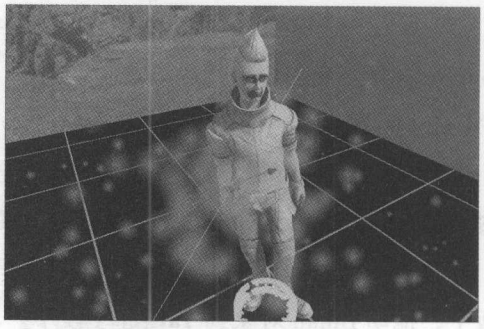
- **Start Size**：0.15
- **Start Color**：挑选一个红色 / 橙色
- **Start Lifetime**：0.3
- **Max Particles**：50

3. 在 **Emission** 中，设置 **Rate**：100。

4. 在 **Shape** 中，设置 **Shape**：**Sphere** 和 **Radius**：0.01。

5. 为了性能，在 **Renderer** 中，设置 **Cast Shadows**：**Off**，**ReceiveShadows**：非选中，**Reflection Probes**：**Off**。

下图是我在 Play 模式下打击 Ethan 胸部时的 **Scene** 视图。



当 Ethan 被射中时，hitEffect 粒子系统被激活。3s 之后（或者任何你设置在 TimeToSelect 变量中的值），他的生命被耗尽，爆炸效果被初始化，得分增加，然后他在一个新的位置重生。在下一章中，我们将展示如何把当前得分展示给玩家。

4.3.3　清理工作

完成前的最后一件事——让我们清理 **Assets** 文件夹，然后将所有的脚本移入 Assets/Scripts/ 子文件夹中。选择 **Project** 的 **Project Assets** 文件夹，创建一个文件夹，命名为 Scripts，把你所有的脚本拖进去。

小结

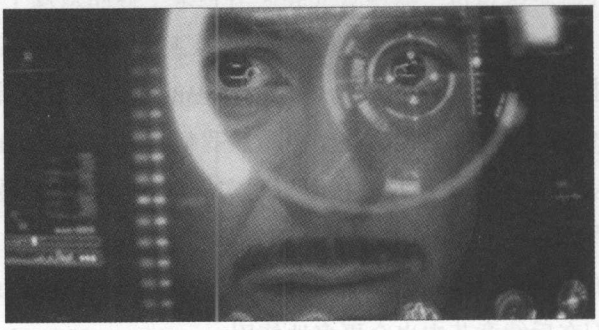
本章中，我们探索了 VR 摄像机和物体在场景中的关系。我们首先让 Ethan 在场景中随机走动，然后使用 NavMesh 让他移动，之后我们再使用一个在 X，Z 平面中的 3D 光标干涉他的流浪。这个光标跟随我们在虚拟现实场景中的视线。最后，我们还使用视线发射了一条射线到 Ethan，让他失血并且最终爆炸。

这些基于视线的技术同样可以应用在非 VR 游戏中，但是在 VR 中，这很平常并且几乎是必需的。我们在后续章节中也会更多地使用。

在下一章中，我们将会着眼于通过多种方法为用户展现信息以及在 VR 中输入小部件。桌面应用和视频游戏中惯用的**用户界面**（UI）在 VR 中可能不适用，因为它们在 Screen Space 中运行。然而，就像我们将要看到的一样，有多种其他有效的方法供你使用，以使虚拟世界中的 UI 更合理。

Chapter 3

第 5 章
世界坐标系 UI



漫威工作室与派拉蒙影业在 2013 年发行的《钢铁侠 3》(图片来源：http://www.geek.com/mobile/wearable-computing-is-back-google-reportedly-making-hud-glasses-1465399/)

图形用户界面（GUI）或者说 UI，通常指的是屏幕上的二维图形，它叠加了主要的游戏界面和用于表示用户当前信息的状态消息、仪表盘，还有类似菜单、按钮、滑动条等的输入控件。

在 Unity 5 中，UI 元素永远都驻留于一个 **canvas** 之上。Unity 手册像下面这样引述了 **canvas** 组件：

> Canvas 组件表示摆放和渲染 UI 的抽象空间。所有 UI 元素都必须是附加了 Canvas 组件的 GameObject 的子对象。

在传统的视频游戏中，UI 对象通常作为一个叠加层渲染在**屏幕空间**（Screen Space）**中的**一个 canvas 之上。屏幕空间的 UI 类似于一块纸板粘在你的电视或显示器上，而游戏操作叠加于其后。

但是，这在虚拟现实中行不通。如果你想在虚拟现实中把屏幕空间用于 UI，你会遇到一些问题。因为虚拟现实中有两个立体的摄像机，你需要为双眼都准备好单独的视图。

而传统的游戏可能会把屏幕的边缘用于 UI，但是虚拟现实并没有屏幕边缘！

在 VR 中，我们使用各种途径把用户界面元素放进**世界坐标系**（World Space）中而不是屏幕坐标系中。本章会描述若干个这些类型，还会详细地定义这些类型并且给一些相同的案例：

- **护目镜平视显示器**（Visor heads-up display）：使用护目镜平视显示器不用考虑你的头部如何移动，用户界面 canvas 始终在你双眼的正前方。
- **十字光标**（Reticle cursor）：与护目镜平视显示器类似，用一个十字或点光标选择场景中的物体。
- **挡风玻璃平视显示器**（Windshield HUD）：一个弹出面板浮动在 3D 空间之中，像是驾驶舱中的挡风玻璃。
- **游戏元素 UI**（Game element UI）：这个 canvas 在屏幕中作为游戏界面的一部分，如同体育馆中的计分板。
- **信息框**（Info bubble）：一个附加到场景中对象上的 UI 消息，像是一个盘旋在角色头部的浮想气泡。
- **游戏中的仪表盘**（In-game dashboard）：一个控制面板，是游戏界面的一部分，高度通常齐腰。
- **可响应的对象 UI**（Responsive object UI）：UI 信息不需要一直出现在视野中，而是根据相关上下文调用它。

这些 UI 技术的不同源于你何时何地显示 canvas，以及用户如何与 canvas 进行交互。本章中，我们将依次试用这些 UI 技术。其中，我们还将继续探讨像点击按钮一样使用头部的移动和姿势进行用户输入。

注意，本章中的有些项目使用第 4 章中完成的场景，但这些项目是独立的，并且不会被本书中的其他章节直接引用。如果你决定跳过其中的某些项目或者不保存它们，也没有关系。

5.1　可重用的默认 canvas

Unity 的 UI canvas 提供了很多的选项和参数，以灵活地适应各种图形布局。不仅期望游戏中具有这些灵活性，还期望网页和移动应用。这些灵活性也带来了复杂性。要想更简单地制作本章中的例子，我们要先构建一个可重用的预制件 canvas，canvas 中有我

们首选的默认设置。

创建一个新的 canvas 并把它的 **Render Mode** 设置成 **world space**，步骤如下：

1. 使用菜单 **GameObject | UI | Canvas**。

2. 把 canvas 重命名为 DefaultCanvas。

3. 把 **Render Mode** 设置成 **world space**。

Rect Transform 组件定义 canvas 自身上面的网格系统，就像坐标图纸上的网格线。它用于在 canvas 中放置 UI 元素。把它设置成方便的 640×480，宽高比为 0.75。**Rect Transform** 组件的宽和高不同于我们场景中 canvas 的世界坐标大小。让我们按照下面的步骤配置 **Rect Transform**：

1. 在 **Rect Transform** 中，设置 **Width** = 640，**Height** = 480。

2. 在 **Scale** 中，把 *X*，*Y*，*Z* 设置成（0.001 35, 0.001 35, 0.001 35）。这是以世界坐标系单位表示的像素大小。

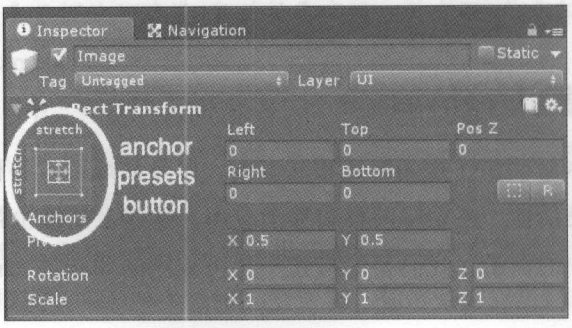
3. 现在，把 canvas 放在距地平面中央一个单位的高度（0.325 是 0.75 的一半）。

在 **Rect Transform** 中，把 **Pos X**、**Pos Y**、**Pos Z** 设置成（0, 1.325, 0）。

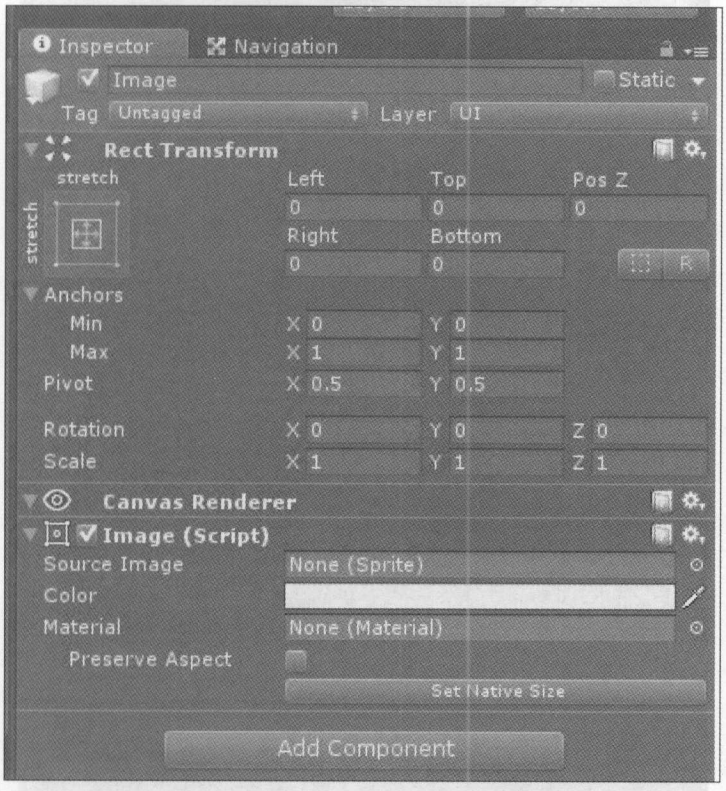
接下来，我们添加一个空的 **Image** 元素（白色背景）以帮助我们将其他透明的 canvas 显示出来，然后为 canvas 制作一个不透明的背景用以不时之需（也可以使用 **Panel** UI 元素）：

1. 选中 DefaultCanvas，点击 **GameObject | UI | Image**（确保其作为 DefaultCanvas 的子对象，如果不是的话，把它移到 DefaultCanvas 之下）。

2. 选中 **Image**，在 **Rect Transform** 面板的左上方，有一个 **anchor presets** 按钮（下图中可以看到）。选择它打开 **anchor presets** 对话框，按住 Alt 键不动查看 **stretch** 和 **position** 选项，然后选择右下角的那个选项（**stretch-stretch**）。现在，拉伸那张（空的）图片以填充 canvas：



3. 对照下图中 DefaultCanvas 子对象 **Image** 默认属性的设置，再次检查 Image 设置。



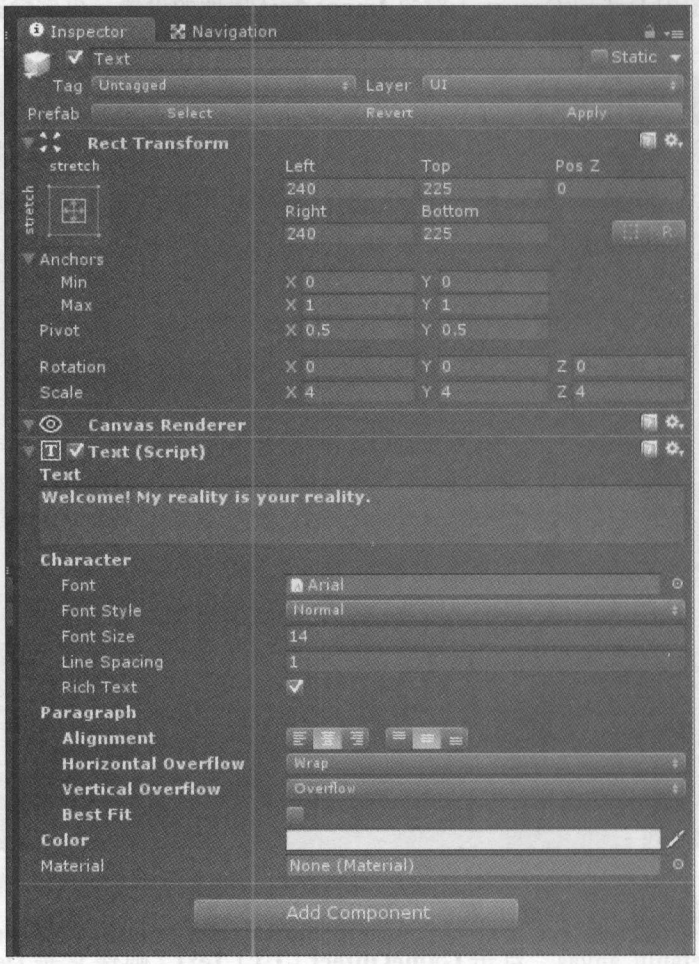
以默认设置添加一个 **Text** 元素，步骤如下：

1. 选中 DefaultCanvas，点击 **GameObject | UI | Text**（确保它是 DefaultCanvas 的子对象，如果不是，把它拖到 DefaultCanvas 之下）。**New Text** 字样应该会出现在 canvas 上。

2. 选中 **Text**，把 **Alignment** 设置成 **Center Align** 和 **Middle Align**，把 **Vertical Overflow** 设置成 **Overflow**，把 **Scale** 设置成（4, 4, 4）。

3. 选中 **Image**，使用 **Rect Transform** 面板左上方的小窗口把其 **anchor presets** 按钮设置成（**stretch-stretch**）。

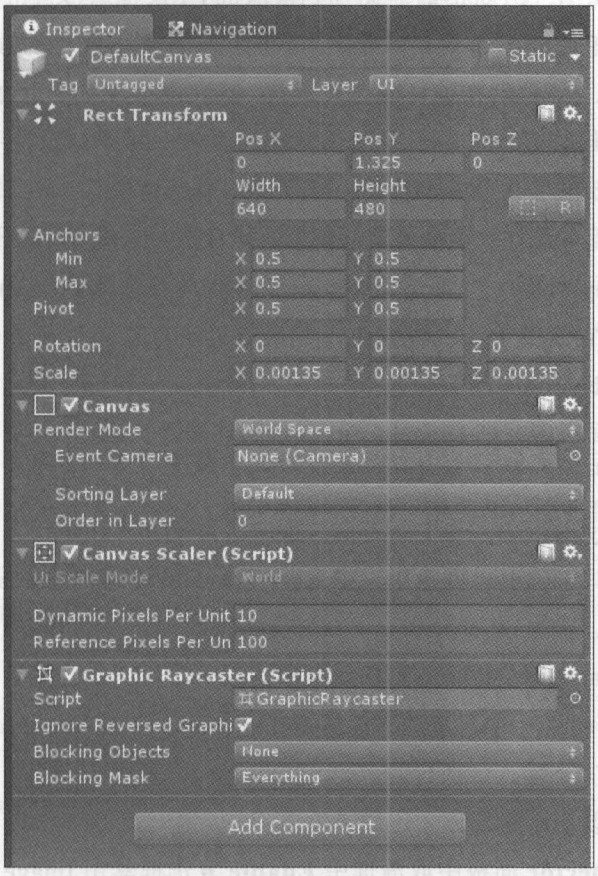
4. 对照下图中 DefaultCanvas 子对象 **Text** 默认属性的设置，再次检查 Text 设置：



保持选中 **DefaultCanvas** 并设置 **Canvas Scaler | Dynamic Pixels Per Unit** 为 10，增加像素的分辨率以便让文本的字体更清晰。

最后，保存成预制件资源，这样就可以在本章中重用了，步骤如下：

1. 根据需要，在 **Project Assets** 中创建一个名为 Prefabs 的文件夹。
2. 把 DefaultCanvas 对象拖进 Project Assets/Prefabs 文件夹下，创建成一个预制件。
3. 现在从 **Hierarchy** 面板中删除 DefaultCanvas 的实例。
4. 对照下图中的属性，再次检查 DefaultCanvas 的设置：



很高兴我们终于明白了。现在，我们使用带有不同 VR 用户界面的 DefaultCanvas 预制件。

canvas 有一个 **Rect Transform** 组件，它定义了 canvas 自身的网格系统，就像坐标图纸上的网格线。它用于摆放 canvas 上的 UI 元素，这与世界坐标系中 canvas 对象的位置和大小不同。

5.2　护目镜 HUD

平视显示器，或者说 HUD，是一个处于你视野中的浮动 canvas，canvas 叠加在游戏界面上。在虚拟现实的术语中，有两个 HUD 的变种，我将这两个变种称为护目镜 HUD（visor HUD）和挡风玻璃 HUD（windshield HUD）。本节讨论第一种。

在护目镜 HUD 中，UI canvas 是附加到摄像机上的。它不会响应你的头部移动，当你的头部移动时，它还是黏附于你的脸上。我们用一种更形象化的方式，假设你戴着一个有护目镜的头盔，UI 映射在护目镜的表面上。这在虚拟现实中也许效果不错，但可能会破坏沉浸感。所以，它应该一般只用于游戏界面的一部分，或者用于带你离开场景，例如用于硬件或系统的工具菜单。

我们用下面的步骤制作一个带有欢迎消息的护目镜 HUD，看看我们自己的感受如何：

1. 在 **Hierarchy** 面板中，展开 MeMyselfEye 结点并钻取到 **Main Camera** 对象（对于 Oculus，我们把它放置于 MeMyselfEye 的最邻近子对象；而对于 Cardboard，你需要进入更深层的 CardboardMain/Head/）。

2. 在 **Project** 面板中，把 DefaultCanvas 预制件拖进 Camera 对象，让它成为 Camera 的子对象。

3. 在 **Hierarchy** 面板中，选中 canvas，重命名为 VisorCanvas。

4. 在 canvas 的 **Inspector** 面板中，把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置为（0, 0, 1）。

5. 展开 VisorCanvas 结点并选择 **Text** 对象。

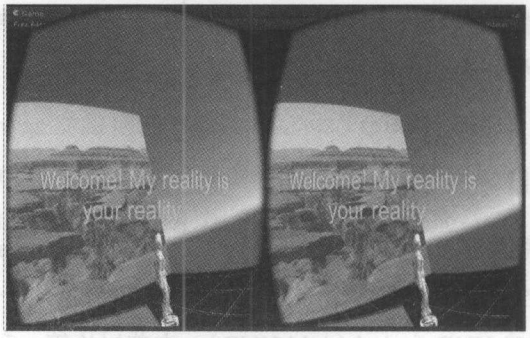
6. 在 **Inspector** 面板中，把文本从 **Default Text** 改成 **Welcome! Mi reality es su reality**。（你可以在文本输入区换行。）

7. 把文本颜色改成某种亮色，比如绿色。

8. 通过在 **Inspector** 面板中取消选中 **Enable** 复选框禁用 Image 对象，使其只显示文本。

9. 保存场景，在 VR 中体验。

以下是使用了 VisorCanvas 的 Rift 屏幕的截图：



在 VR 中，当你向四周移动头部时，文本会随之移动，因为它附加到了你的脸部正前方的护目镜上。

护目镜 HUD 和十字光标 canvas 被设置成了摄像机的子对象。

现在，要么禁用 VisorCanvas，要么删除它（在 **Hierarchy** 面板中，点击右键选择 **Delete**），因为我们将要在后面的章节中用不同的方式显示欢迎消息。接下来，我们讨论这项技术的一个其他应用。

5.3　十字光标

护目镜 HUD 的一种变体是：在第一人称射击游戏中十字线或十字光标是一个必需品。类似你正看着瞄准镜或者显微镜（而不是护目镜），而且你头部的移动与枪或炮塔本身一起移动。你可以用一个普通的游戏对象制作它（比如一个四边形 + 一张纹理图片），但是本章关于 UI，所以使用 canvas，步骤如下：

1. 在 **Hierarchy** 面板中找到 **Main Camera** 对象。

2. 在 **Project** 面板中，把 DefaultCanvas 预制件拖进 camera 对象，让它成为 camera 的子对象，把它命名为 ReticleCursor。

3. 把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置为（0, 0, 1）。

4. 删除其子对象——**Image** 和 **Text**。

5. 通过在主菜单栏点击 **GameObject | UI | Raw Image** 添加一张原始图片，确保它是 ReticleCursor 的子对象。

6. 在 **Raw Image** 面板的 **Rect Transform** 中，把 **Pos X**、**Pos Y**、**Pos Z** 值设置为（0, 0, 0），**Width** 和 **Height** 设置成（22, 22）。然后，选择一个显眼的 **Color**，比如 **Raw Image(Script)** 的红色。

7. 保存场景，在 VR 中体验。

如果你喜欢一种更好看的十字线，在 **Raw Image(Script)** 属性中，把 **Texture** 字段换成一张光标图片。比如，点击 **Texture** 字段最右边的小圆图标，打开 **Select Texture** 对话框，找到并选择一个适合的十字线，比如 Crosshair 图片。（本书中有 Crosshair.gif 的副本。）确认 **Width** 和 **Height** 的值设置成所选图片大小（Crosshair.gif 的大小是 22×22），确认 **Anchor** 被设置成 **middle-center**。

把 **Pos Z** 设置成 1.0，这样十字光标就浮动在距离你前方 1m 的位置。一个固定距离的光标在大多数 UI 情况下表现良好。例如，当你从一块平的 canvas 中拾取某个物体时，还是一个固定的距离。

然而，这是世界坐标系。如果另一个物体在你和十字线之间，十字线会变模糊。

另外，如果你看很远的物体，你将重新聚焦你的眼睛，并且在同时观察光标时会有问题。(为了强调这个问题，试着把光标移得更近一些。例如，如果你把 ReticleCursor 的 **Pos Z** 设置成 0.5 或更小，你可能需要斜视才能看见它！) 要解决这个问题，我们可以投射光线并把光标移动到你观察物体的实际距离，相应改变光标的大小让它保持相同的大小。下面是一个比较方便的方案：

1. 选中 ReticleCurosr，点击 **Add Component | New Script**，命名为 CursorPositioner，并点击 **Create** 和 **Add**。

2. 通过双击它在 MonoDevelop 中打开这个脚本。

这是 CursorPositioner.cs 脚本的代码：

```
using UnityEngine;
using UnityEngine.EventSystems;
using System.Collections;

public class CursorPositioner : MonoBehaviour {
  private float defaultPosZ;

  void Start () {
    defaultPosZ = transform.localPosition.z;
  }

  void Update () {
    Transform camera = Camera.main.transform;
    Ray ray = new Ray (camera.position, camera.rotation *
      Vector3.forward);
    RaycastHit hit;
    if (Physics.Raycast (ray, out hit)) {
      if (hit.distance <= defaultPosZ) {
        transform.localPosition = new Vector3(0, 0, hit.distance);
      } else {
        transform.localPosition = new Vector3(0, 0, defaultPosZ);
      }
    }
  }
}
```

脚本中的 `transform.localPosition` 是 **Rect Transform** 组件的 **Pos Z** 的值，如果它比给出的 **Pos Z** 的值小，脚本会把这个值改成 `hit.distance`。现在，你可以把十字线移动到一个更舒服的距离上，比如 **Pos Z** = 2。

@eVRydayVR 有一个非常不错的教程，展示了如何实现距离和大小补偿世界坐标系间中的十字线。你可以访问 https://www.youtube.com/watch?v=LLKYbwNnKDg 观看标题为 *Oculus Rift DK2 – Unity Tutorial: Reticle* 的视频。

我们刚才实现了我们自己的十字光标，而现在很多虚拟现实 SDK 也提供了光标（对于 Oculus Rift，可查看 OVRCorsshair.cs；对于 Cardboard，它是 GazeInputModule.cs 工具的一部分），虽然它们不一定有距离和大小的补偿。

5.4　挡风玻璃 HUD

术语平视显示器（Heads-up Display，HUD）原用于飞机。飞行员可以向前平视头部姿势而不用低头去查看仪器面板上的信息。由于这个用途，我们称之为挡风玻璃 HUD（windshield HUD）。就像护目镜 HUD 一样，信息面板叠加于游戏界面之上，但它并不附着于你的头部。反而，你可以认为它附着在驾驶座舱或牙科诊所的座位上。

护目镜 HUD 就像是 UI canvas——它附着在你的头部，而挡风玻璃 HUD 像是附着在你的座椅上。

我们用下面的步骤创建一个简单的挡风玻璃 HUD：

1. 在 **Project** 面板中，把 DefaultCanvas 预制件拖到 **Hierarchy** 面板中的 MeMyselfEye 对象之下，让它成为 MeMyselfEye 的子对象。

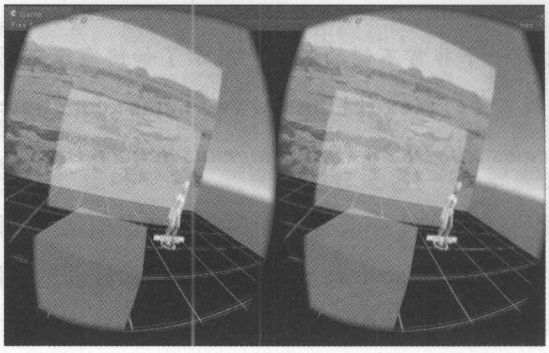
2. 将其命名为 HUDCanvas。

3. 选中 HUDCanvas，把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 设置成（0, 0.4, 0.8）。

4. 现在，设置 **Text** 组件。选中 HUDCanvas 下的 **Text**，将文本改成 **Welcome! Mi reality es su reality**。另外，把颜色也改成某种亮色，比如绿色。

5. 这一次，我们把面板变成透明的。选中 HUDCanvas 下的 **Image**，选中其调色板，然后在 Color 对话框中，把 **Alpha**（“A”）通道从 255 改成 115。

相当简单，当你在 VR 中观察它时，canvas 就在你前方；但是如果你向四周看，它的位置似乎仍然是静止的，而且是相对于场景中的其他物体，如下面的截图所示。



在第 6 章中，我们会知道，当一个第一人称角色在场景中穿过时，HUD canvas 还会在你的前方，在与相对于你的身体对象 MeMyselfEye 的相同位置上。

你也许意识到场景中的物体可能会使 HUD canvas 变模糊，因为它们都占据了相同的世界坐标系。如果你想要避免这种情况，你需要保证 canvas 永远是最后渲染的，这样它就出现在所有其他对象的前面了，而不用关心 3D 空间的位置。在一个传统的单视场（monoscopic）游戏中，你可以通过为 UI 添加第二个摄像机和改变其渲染优先级做到这一点。在立体 VR（stereoscopic VR）中，你必须用不同的方式实现这一点，可能需要为你的 UI 对象写一个自定义的着色器（shader）(一个进阶话题)。

这种 HUD 的一个变种是转动 canvas，让它永远面对着你，这样它在 3D 空间中的位置就是固定的了。参见 5.6 节学习如何用代码实现。

为了好玩，我们写一个脚本在 15s 后移除欢迎消息的 canvas，步骤如下：

1. 选中 HUDCanvas，点击 **Add Component | New Script**，把脚本命名为 DestroyTimeout，再点击 **Create** 和 **Add**。

2. 在 MonoDevelop 中打开这个脚本。

DestroyTimeout.cs 脚本如下：

```
using UnityEngine;
using System.Collections;

public class DestroyTimeout : MonoBehaviour {
  public float timer = 15.0f;
```

```
    void Start () {
      Destroy (gameObject, timer);
    }
}
```

当游戏开始后，HUD Canvas 将会在计时器到期后消失。

挡风玻璃 HUD canvas 被设置成第一人称人物角色的子对象，作为摄像机的兄弟对象。

这个例子中，我们开始尝试第一人称角色。想象坐在一个汽车中或飞机的驾驶舱内，HUD 映射在你前方的挡风琉璃上，而你可以自由地移动你的头部向四周看。在场景中的 **Hierarchy** 面板中，有一个第一人称角色对象（MeMyselfEye），其中包括摄像机以及可能的虚拟角色的身体和你周围的其他装备。当游戏中的车辆移动时，整个座舱会一起移动，包括摄像机和挡风玻璃。我们稍后会在本章后文和第 6 章中继续讨论。

5.5　游戏元素 UI

当 Ethan 被杀害时，GameContoller 对象的 KillTarget 脚本中的得分值会更新，但是我们没有把当前得分显示给玩家（上一章设置的）。我们现在来显示它——添加一个得分板到场景中的背景图片 PhotoPlane 的左上角：

1. 在 **Project** 面板中，把 DefaultCanvas 预制件直接放进 **Scene** 视图。

2. 将其命名为 ScoreBoard。

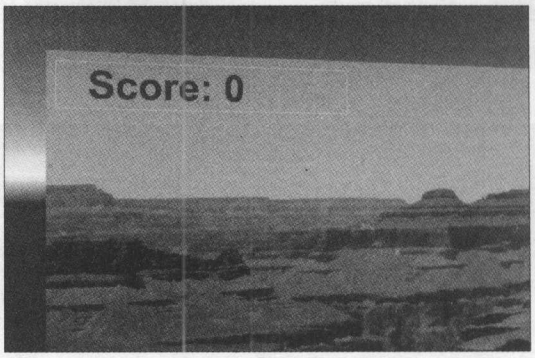
3. 选中 ScoreBoard，把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置成（-2.8, 7, 4.9），**Width** 和 **Height** 设置成（3 000, 480）。

4. 选中 ScoreBoard 下的 **Text**，把 **Font Size** 设置成 100，并把 **Text** 的颜色设置成一种显眼的颜色，比如红色。

5. 在 **Text** 中输入 **Score: 0**，作为示例字符串。

6. 通过反选 **Enable** 复选框或删除此 Image 以禁用 ScoreBoard 下的 **Image**。

我们添加另一个 canvas 到场景中，大小、位置和文本均由我们自己决定，格式化文本用于显示，如下图这样：



现在，我们需要更新 KillTarget.cs 脚本，像下面这样：

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class KillTarget : MonoBehaviour {
  public GameObject target;
  public ParticleSystem hitEffect;
  public GameObject killEffect;
  public float timeToSelect = 3.0f;
  public int score;
  public Text scoreText;

  private float countDown;

  void Start () {
    score = 0;
    countDown = timeToSelect;
    hitEffect.enableEmission = false;
    scoreText.text = "Score: 0";
  }

  void Update () {
    Transform camera = Camera.main.transform;
    Ray ray = new Ray (camera.position, camera.rotation *
      Vector3.forward);
    RaycastHit hit;
    if (Physics.Raycast (ray, out hit) && (hit.collider.gameObject
      == target)) {
      if (countDown > 0.0f) {
        // on target
        countDown -= Time.deltaTime;
        // print (countDown);
```

```
            hitEffect.transform.position = hit.point;
            hitEffect.enableEmission = true;
          } else {
            // killed
            Instantiate( killEffect, target.transform.position,
              target.transform.rotation );
            score += 1;
            scoreText.text = "Score: " + score;
            countDown = timeToSelect;
            SetRandomPosition();
          }
        } else {
          // reset
          countDown = timeToSelect;
          hitEffect.enableEmission = false;
        }
      }

      void SetRandomPosition() {
        float x = Random.Range (-5.0f, 5.0f);
        float z = Random.Range (-5.0f, 5.0f);
        target.transform.position = new Vector3 (x, 0.0f, z);
      }
    }
```

保存脚本后，回到 Unity 编辑器，在 **Hierarchy** 面板中选择 GameController，然后从 **Hierarchy** 面板中把 ScoreBoard 下的 **Text** 对象拖到 **Kill Target（Script）**中的 **Score Text** 字段中。

在虚拟现实中运行场景，每次你杀死 Ethan（通过注视他）后得分会在 PhotoPlane 左上角的 ScoreBoard 中更新。

一个游戏元素 UI canvas 和其他游戏对象一样是场景的一部分。

这个例子是把场景中的一个对象用来显示信息。我们的例子相当简单，你可能想要制作一个更好的模块化的得分板，就像在体育馆或某些地方中看到的那种。关键是，它是场景的一部分，要看到这个得分消息你可能得转动你的头部，好吧，看着它。

5.6　信息框

在漫画书中，当角色说话时内容会显示在一个对话框中。在很多在线社交世界中，

参与者都表现为虚拟头像，然后如果你（把鼠标）停在某个人的头像上，他们的名称会显示出来，我把这种类型的UI称作信息框。

信息框放在世界坐标系中的某个特定的3D位置上，但是canvas应该永远都面向摄像机。我们可以用脚本确保这一点。

在这个例子中，我们显示WalkTarget对象的***X***和***Z***位置（上一章设置的），通过LookMoveTo.cs脚本控制。用下面的步骤添加信息框。

1. 在**Project**面板中，把DefaultCanvas预制件直接拖进**Scene**视图中，放在GameController下面的WalkTarget的顶部，让它作为WalkTarget的子对象。

2. 将其命名为InfoBubble。

3. 选中InfoBubble，把**Rect Transform**组件的**Pos X**、**Pos Y**、**Pos Z**值设置成（0, 0.2, 0）。

4. 选中InfoBubble下的**Text**，把**Rect Transform**组件的**Pos X**、**Pos Y**、**Pos Z**值设置成（0, 0, 0），**Right**和**Bottom**设置成（0, 0）。

5. 选中InfoBubble下的**Image**，将其**Scale**设置成（0.7, 0.2, 1）。

6. 在**Text**中输入**X:00.00**，**Z:00.00**作为示例字符串。

粗略地看一下canvas和文本的大小和位置，并按你的想法调整文本。

现在，我们要修改LookMoveTo.cs脚本以显示当前的WalkTarget的***X***和***Z***位置。在MonoDevelop编辑器打开此脚本，添加下面的代码：

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class LookMoveTo : MonoBehaviour {
  public GameObject ground;
  public Transform infoBubble;

  private Text infoText;

  void Start () {
    if (infoBubble != null) {
      infoText = infoBubble.Find ("Text").GetComponent<Text> ();
    }
  }

  void Update () {
```

```
    Transform camera = Camera.main.transform;
    Ray ray;
    RaycastHit[] hits;
    GameObject hitObject;

    ray = new Ray (camera.position, camera.rotation *
      Vector3.forward);
    hits = Physics.RaycastAll (ray);
    for (int i=0; i < hits.Length; i++) {
      RaycastHit hit = hits [i];
      hitObject = hit.collider.gameObject;
      if (hitObject == ground) {
        if (infoBubble != null) {
          infoText.text = "X:" + hit.point.x.ToString("F2") + ",
            Z:" + hit.point.z.ToString("F2");

          infoBubble.LookAt(camera.position);
          infoBubble.Rotate ( 0.0f, 180.0f, 0.0f );
        }
        transform.position = hit.point;
      }
    }
  }
}
```

using UntiyEngine.UI; 这行代码表示此脚本需要访问 Unity 的 UI API。我们定义了一个 Public Transform infoBubble 变量，将其设置给 WalkTarget/InfoBubble 对象。我们还定义了一个 private 的 **Text**，infoText 变量，值被设置成了 InfoBubble 对象的 **Text** 对象。此脚本假设给出的 InfoBubble 有一个子 **Text** UI 对象。

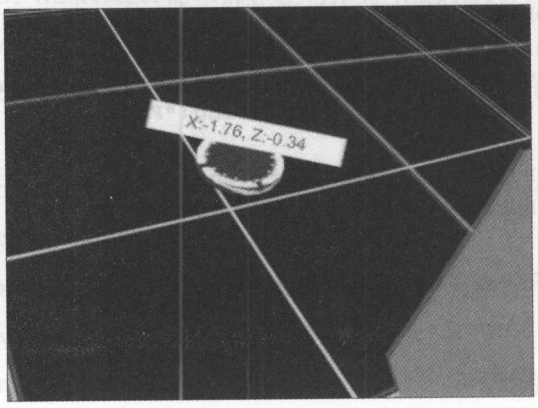
这样 infoText 文本对象就有了一个文本组件，有一个文本字符属性！你可以在 Unity 编辑器中看见我指的是什么。如果你在选中 InfoBubble/Text 时检查 **Inspector** 面板，你会发现它有一个 **Text（Script）**组件，其中有一个 **Text** 字段，这个 **Text** 字段就是我们写消息的地方。所以，在 Setup() 中，我们发现 WalkTarget/InfoBubble/Text 对象把 **Text** 对象赋值给 infoText，而在 Update() 中，我们设置 infoText.text 的字符值让得分显示在 bubble canvas 中。

另外，在 Update() 中，我们使用 infoBubble.LookAt() 并给它传入摄像机的位置以变换 infoBubble canvas 使它始终面对我们，LookAt() 让 canvas 始终面对我们。所以，我们还需要让它绕着 y 轴旋转 180° 。

保存脚本，并把 **InfoBubble** 对象从 **Hierarchy** 拖进 **LookMoveTo(script)** 组件的 **InfoBubble** 槽中。如果你不为 `InfoBubble canvas` 赋值，脚本仍然会运行，因为我们在引用它之前测试了 `null` 对象。

一个信息框 UI canvas 是附着在其他游戏对象上的，随着游戏对象移动并且始终面对着摄像机（就像一块广告牌）。

在 VR 中运行场景，你会看到 WalkTarget 有一个小的信息框告诉我们其 *X* 和 *Z* 的位置。



额外挑战： 想要试一些其他的功能吗？给 Ethan 实现一个生命值条。使用 KillTarget 脚本中的 countDown 变量检查其父对象的生命值，然后当它的生命值不是 100 时在它的头部上方显示生命值（横条）。

当你需要显示属于场景中特定对象的 UI 消息并且 UI 消息与对象一起移动时，InfoBubble 很有用。

5.7 响应输入事件的游戏内仪表板

游戏内的仪表板是一个集成进游戏本身的 UI 界面。典型的场景是你坐在一辆汽车或一艘太空船的座舱中，在腰部高度（桌椅高度）上是一个包括一组控制器、仪表、信息显示栏等的面板。仪表板通常在坐下体验虚拟现实时感觉起来更自然。

前几页中，我们讨论了 windshield HUD。仪表板也差不多是一样的，有一处不同是仪表板可能更明显地是关卡中的一部分，而不是简单地作为一个附加信息或菜单。

事实上，仪表板可以作为一个非常有效的装置来控制 VR 产生的晕动症。研究人员

已经发现，当 VR 用户有一个更好的着地感并在视野中有一条不变的地平线时，在虚拟空间中向四周移动时能大大减少头晕感。相反，浮动的一维眼球没有自我感和着地感是自找麻烦。（参见 Oculus Best Practices 的说明和其他一些很棒的技巧 https://developer.oculus.com/documentation/intro-vr/latest/concepts/bp_intro/ 。）

在这个例子中，我们要用一个“开始 / 结束”按钮制作一个简单的仪表板。目前，按钮将会操作场景中的一个水管来避开僵尸。（不错吧？）这个项目使用上一章创建的场景。

这个项目可能比你想的要复杂一点儿，但是，如果你已经在 Minecraft 中构建过东西的话，你就会知道即使很简单的东西也需要组装不同的部件。下面是我们要做的：

1. 创建一个带有两个功能按钮的仪表板 canvas——“开始”和“结束”。
2. 添加一个水管到场景中，连接两个按钮。
3. 写一个简单的脚本激活这两个按钮。
4. 通过注视按钮来高亮它。
5. 改进脚本，使只有在它被高亮后才激活这个按钮。
6. 修改十字光标（之前创建的），让它只有在注视仪表板时才启用。
7. 考虑使用不同 VR 硬件时的 click 机制。

让我们这样做吧。

5.7.1　用按钮创建仪表板

首先，我们创建一个带有“开始”和“结束”两个按钮的仪表板，步骤如下：

1. 在 **Project** 面板中，把 DefaultCanvas 预制件拖进 **Hierarchy** 面板中的 MeMyselfEye 对象中，成为其子对象。
2. 重命名为 Dashboard。
3. 选中 Dashboard，把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置成（0, -1, 0.2），**Rotation** 设置成（60, 0, 0）。
4. 禁用或删除 Dashboard 的 **Text** 子对象。

这是把仪表板放在我们的双眼之下 1m 的位置，且稍微前面靠外一点。

看一下这个半成品，如果你喜欢，我已经包含一个车的仪表盘的图片集供你使用，步骤如下：

1. 把 DashboardSketch.png 文件导入 **Project**（比如 Assets/Textures 文件夹）。
2. 添加一个新的 **GameObejct | UI | Raw Image**，作为 Dashboard 的子对象。
3. 把 DashboardSketch 纹理从 **Project** 面板拖进 **Raw Image** 组件的 **Texture** 字段。

.4. 把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置成（0, 0, 0）、**Width** 设置成 140，**Height** 设置成 105 。

5. 它应该是 **Anchored** 在 ***X*** 、***Y*** 和 **Pivot** 的值为（0.5, 0.5）、**Rotation**（0, 0, 0）。

6. 把 **Scale** 设置成（4.5, 4.5, 4.5）。

接下来，我们添加“开始”和“停止”按钮。它们可以放在 canvas 中任何你想放的地方，但是草图中有两个不错的位置预留给它们了：

1. 添加一个新的 **GameObject | UI | Button** 作为 Dashboard 的子对象，命名为 StartButton。

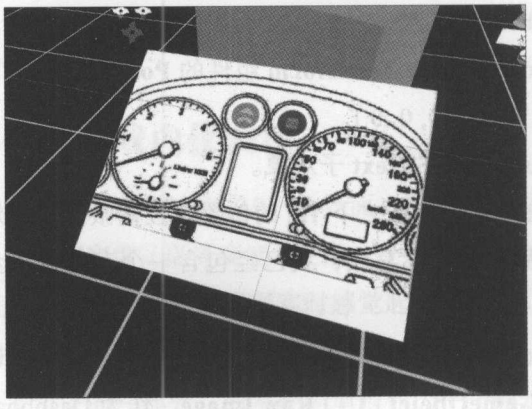
2. 把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置成（–48, 117, 0），**Width** 和 **Height** 设置成（60, 60），**锚定**（Anchored）在中心（0.5），没有 **Rotation** 而 **Scale** 是 1。

3. 在按钮的 **Image（Script）** 组件面板中，对于 **Source Image**，点击右边的小圆圈以打开 **Select Sprite** 拾取并选择 **ButtonAcceleratorUpSprite**（你也许已经导进了 Assets/Standard Assets/CrossPlatformInput/Sprites 文件夹）。

4. 在按钮的 **Button（Script）** 组件面板中，对于 **Normal Color**，我设置成 RGB（89, 154, 43），而 **Highlighted Color** 设置成（105, 225, 0）。

5. 类似地，创建另一个按钮并命名为 **StopButton**，把 **Rect Transform** 组件的 **Pos X**、**Pos Y**、**Pos Z** 值设置成（52, 118, 0），**Width** 和 **Height** 设置成（60, 60），**Source Image** 选择 **ButtonBrakeOverSprite**，**Normal Color** 设置成（236, 141, 141），**Highlighted Color** 设置成（235, 45, 0）。

结果应该看起来像这样：



2001 奥迪 A6（源自 http://www.justanswer.com/audi/6sskf-audi-a6-audi-a6-51-regcentre-dashboard.html#re.v/174/）

最后一件事，如果你正在使用本章之前用 CursorPositioner.cs 脚本创建的 ReticleCursor，我们想让仪表板自己有一个碰撞器。我们可以通过下面的步骤达到目的：

1. 选中 Dashboard，右键选择 **3D Object | Pane**。

2. 把 **Position** 设置成（0, 0, 0），**Rotation** 设置成（270, 0, 0），**Scale** 设置成（64, 1, 48）。

3. 禁用 **Mesh Renderer**（但是保持启用 **Mesh Collider**）。

现在仪表板有一个没有渲染的平面子对象，但是当 CursorPositioner 做光线投射时它的碰撞器会被检测到。

用一个带有“按压”和“松开”状态的开关按钮也许要比分开的两个“开始”和“结束”按钮要好。我们完成这个之后，去理解一下是怎么做的。

我们刚刚创建了一个世界坐标系中的 canvas，它应该出现在虚拟现实中的腰部或桌子的高度。我们用一个仪表板草图和两个 UI 按钮装饰它。现在，我们将给这两个按钮连接上特殊的事件。

5.7.2　连接水管与按钮

我们先给这些按钮添加一些操作，比如打开消防水管的动作。如果我们从战略上瞄准它，它甚至可以防守凶猛的僵尸。巧合的是，之前在 Unity 中导入的 **Standard Assets** 下的 **Particle Systems** 有一个水管，我们可以利用。把它加入场景中，步骤如下：

1. 如果你之前没有导入的话，那么通过主菜单栏的 **Assets | Import Package | ParticleSystems** 导入 **Particle Systems** 标准资源。

2. 在 **Porject** 面板中，找到 Assets/Standard Assets/ParticleSystems/Prefabs/Hose 预制件，把它拖进 **Scene** 视图中。

3. 把 **Transform** 组件的 *X*, *Y*, *Z* 设置成（−3, 0, 1.5），把 **Rotation** 设置成（340, 87, 0）。

4. 确保 **Hose** 呈启用状态（查看 **Enable** 复选框）。

5. 打开 **Hierarchy** 中的 **Hose**，这样你就可以看见它的子对象 **WaterShower** 粒子系统，选择它。

6. 在 **Inspector** 中的 **Particle System** 属性面板中，查看 **Play On Awake** 并反选之。

注意 **Hierarchy** 中的 **Hose** 对象有一个 **WaterShower** 的子对象。这是我们将要用按钮控制的真正的粒子系统，它起初应该是关闭状态。

Hose 预制件其本身是由鼠标控制的脚本，我们不需要用它，所以禁用之，步骤如下：

1. 选中 **Hose**，禁用（反选）它的 **Hose（Script）**。

2. 禁用（反选）**Simple Mouse Rotator（Script）**组件。

现在，我们要把 StartButton 连接到 **WaterShower** 粒子系统上，通过让按钮监听 OnClick( ) 事件，步骤如下：

1. 打开 **Hierarchy** 中的 **Hose**，这样你就可以看见它的子对象 **WaterShower** 粒子系统。

2. 在 **Hierarchy** 中，选择 StartButton（在 MeMyselfEye/Dashboard 下）。

3. 注意 **Inspector** 中的 **OnClick( )** 为空。点击面板右下角的加号（+）图标让它显示一个标签为 **None（Object）** 新的字段。

4. 把 **WaterShower** 粒子系统从 **Hierarchy** 拖进 **None（Object）**字段。

5. 它的函数选择器的默认值是 **No Function**。把它改成 **ParticleSystem | Play( )**。

好了，StopButton 的步骤也差不多，如下：

1. 在 **Hierarchy** 中，选择 StopButton。

2. 点击 **OnClick()** 面板右下角的加号（+）。

3. 把 **WaterShower** 粒子系统从 **Hierarchy** 拖进 **None（Object）** 字段。

4. 它的函数选择器的默认值是 **No Function**。把它改成 **ParticleSystem | Stop( )**。

开始和停止按钮监听了 OnClick( ) 事件，当有事件时，它会相应地调用 WaterShower 粒子系统的 Play( ) 和 Stop( ) 函数。我们需要按下按钮来完成这个操作。

5.7.3 用脚本激活按钮

在我们给用户提供按下按钮的途径之前，我们看一下如何才能用脚本来完成。在 GameController 上创建一个脚本，步骤如下：

1. 选中 GameController，按下 **Add Componenet | New Script**，创建一个命名为 ButtonExecuteTest 的脚本。

2. 在 MonoDevelop 中打开脚本。

在下面的脚本中，我们每 5s 间隔关闭一次管子，如下：

```
using UnityEngine;
using UnityEngine.EventSystems;
using System.Collections;

public class ButtonExecuteTest : MonoBehaviour {
  private GameObject startButton, stopButton;
  private bool on = false;
  private float timer = 5.0f;

  void Start () {
    startButton = GameObject.Find ("StartButton");
    stopButton = GameObject.Find ("StopButton");
  }

  void Update () {
    timer -= Time.deltaTime;
    if (timer < 0.0f) {
      on = !on;
      timer = 5.0f;

      PointerEventData data = new PointerEventData
        (EventSystem.current);
      if (on) {
        ExecuteEvents.Execute<IPointerClickHandler> (startButton,
          data, ExecuteEvents.pointerClickHandler);
      } else {
        ExecuteEvents.Execute<IPointerClickHandler> (stopButton,
          data, ExecuteEvents.pointerClickHandler);
      }
    }
  }
}
```

脚本管理一个布尔值，这个值表示管子是打开的还是关闭的。它有一个计时器可以在每次 update 后从 5s 开始倒计时。我们对变量使用 `private` 关键字，让它只能用于此脚本内部，而 `public` 关键字声明的变量可以被 Unity 编辑器和其他脚本访问和修改。对于 `startButton` 和 `stopButton`，我决定用 `GameObject.Find()` 获取它们，而不是在 Unity 编辑器中使用拖放的形式。

在这个脚本中，我们引入了 UnityEngine.EventSystem。Events 是一种为不同组件间提供交互的方式。当事件发生时，比如按下按钮，另一个脚本的函数可能会被调用。在这里，我们触发一个事件以对应按下开始按钮，触发另一个以对应按下结束按钮。

我们脚本中的核心是ExecuteEvents.Execute函数，因为我们设置让按钮响应OnClick()事件，我们只需要发送一个事件给这个按钮。当我们想打开管子时，我们调用ExecuteEvents.Execute<IPointerClickHandler> (startButton, data, ExecuteEvents.pointerClickHandler)。当我们想要关闭管子时，我们也为stopButton调用这个函数。这个API需要包含一个PointerEventData对象参数。所以，我们也提供了。(更多关于Unity事件脚本的信息，参见http://docs.unity3d.com/Manual/EventSystem.html。)

保存脚本，点击Play。管子应该打开然后每5s关闭一次（就像是一只小猫行走在仪表板上）。

现在我们测试点击按钮和管子与事件系统的连接，我们可以在继续之前禁用脚本。

把一个复杂的功能分解成很小的功能，再分别测试它们，是一种优秀的实现策略。

5.7.4 用注视高亮显示按钮

同时，让我们检测何时用户正注视着一个按钮，然后高亮显示它。

尽管Button是一个Unity的UI对象，但是它需要被光线投射检测到。可能还有其他方法可以实现这一点，但是在这里我们要给每个按钮添加一个球体游戏对象，并把tag标记为Button，让它投射出一束光线去检测它。首先，通过下面的步骤添加一个球体：

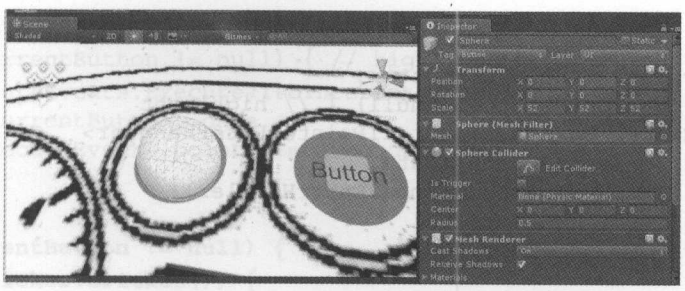
1. 在**Hierarchy**面板中，选择StartButton（MeMyselfEye/Dashboard下），右键选择**3D Object | Sphere**。

2. 把**Transform**组件的**Scale**设置成（52, 52, 52），让它适合按钮的大小。

3. 在**Inspector**面板的顶部，点击**Tag**选择器，点击**Add Tag**，然后再选择加号（+）图标在标签列表中添加一行。输入Button以替代**New Tag**。

4. 我们现在标记StartButton的子对象球体。所以，再次点击在**Hierarchy**面板中的球体，让它出现在**Inspector**面板中。然后，点击**Tag**选择器。这次，Button应该在列表中，选择它。

下面的截图显示了**Scene**中的按钮上的球体，我们禁用它的**Mesh Renderer**之前在**Inspector**中将它标记为Button：



通过反选 **Mesh Renderer** 复选框以禁用球体的 **Mesh Renderer**。

再次对 StopButton 重复这几步操作。

现在，给 GameController 创建一个新的脚本，步骤如下：

1. 选中 GameController，如果还没有的话，现在通过反选它的 **Enable** 复选框禁用之前的 ButtonExecuteTest 脚本。

2. 选中 GameController，点击 **Add Componenet | New Script** 以创建一个命名为 ButtonExecute 的脚本。

3. 在 MonoDevelop 中打开脚本。

在下面的 ButtonExecute.cs 脚本中，我们让按钮在你注视它时高亮显示：

```
using UnityEngine;
using UnityEngine.EventSystems;
using System.Collections;

public class ButtonExecute : MonoBehaviour {
  private GameObject currentButton;

  void Update () {
    Transform camera = Camera.main.transform;
    Ray ray = new Ray(camera.position, camera.rotation *
      Vector3.forward );
    RaycastHit hit;
    GameObject hitButton = null;
    PointerEventData data = new PointerEventData
      (EventSystem.current);
    if (Physics.Raycast (ray, out hit)) {
      if (hit.transform.gameObject.tag == "Button") {
        hitButton = hit.transform.parent.gameObject;
      }
    }
    if (currentButton != hitButton) {
      if (currentButton != null) { // unhighlight
        ExecuteEvents.Execute<IPointerExitHandler> (currentButton,
```

```
            data, ExecuteEvents.pointerExitHandler);
        }
        currentButton = hitButton;
        if (currentButton != null) { // highlight
          ExecuteEvents.Execute<IPointerEnterHandler>
           (currentButton, data,
           ExecuteEvents.pointerEnterHandler);
        }
      }
    }
}
```

对你来说，这个脚本的大部分代码现在看起来应该似曾相识。在每帧的 Update() 里，我们从摄像机投射了一束光线，通过光线碰撞寻找一个标记为 Button 的对象（按钮的球体）。那么，我们把它的父对象（UI 按钮自己）声明为 hitButton。我们记得每次调用 Update() 时高亮的 currentButton。那么，我们可以检测新的光线碰撞是否相同（或根本没有，为空）。

此脚本在 Button 对象上执行两个新的事件。在之前的 ButtonExecuteTest.cs 脚本中，我们执行了一个 click 事件（pointerClickHandler），这次，我们将执行 enter 和 exit 事件对应高亮和移除高亮（pointerEnterHandler 和 pointerExitHandler）。

保存脚本并运行。当你注视一个按钮时，它应该会高亮显示；而当你视线离开它时，它会移除高亮显示。

5.7.5 注视并点击选择

要做一个功能性的仪表板，按钮应该在被点击时起作用。我们使用第 3 章中写的通用 Clicker 类，它检测用户在键盘、鼠标或 Cardboard 触发器上的点击。

在 ButtonExecute.cs 脚本的顶部的类定义中，添加以下代码：

```
public class ButtonExecute : MonoBehaviour {
  private GameObject currentButton;
  private clicker = new Clicker ();
```

然后修改 ButtonExecute.cs 脚本也很简单。在 Update() 的底部，做以下修改：

```
...
if (currentButton != hitButton) {
  if (currentButton != null) { // unhighlight
    ExecuteEvents.Execute<IPointerExitHandler> (currentButton,
      data, ExecuteEvents.pointerExitHandler);
  }
```

```
    currentButton = hitButton;
    if (currentButton != null) { // highlight
      ExecuteEvents.Execute<IPointerEnterHandler>
        (currentButton, data,
        ExecuteEvents.pointerEnterHandler);
    }
  }
  if (currentButton != null) {
    if (clicker.clicked()) {
      ExecuteEvents.Execute<IPointerClickHandler>
        (currentButton, data,
        ExecuteEvents.pointerClickHandler);
    }
  }
```

设置高亮后，如果还有一个高亮的并且按下了任意键（键盘或鼠标），我们就把它当作一次选择并点击在 UI 按钮上。

我们现在有一个带有按钮的游戏内仪表板可以响应用户输入，用于控制场景中某个对象（水管）的行为。

5.7.6　注视并聚焦选择

不使用 clicker 的话，我们还可以用一个基于时间的选择用于点击按钮。要做到这一点，我们在注视一个按钮时使用一个倒计时器，更像是我们在上一章中用来杀死 Ethan 的那个。ButtonExecute.cs 脚本大致如下：

```
using UnityEngine;
using UnityEngine.EventSystems;
using System.Collections;

public class ButtonExecute : MonoBehaviour {
  public float timeToSelect = 2.0f;
  private float countDown;
  private GameObject currentButton;
  private clicker = new Clicker ();

  void Update () {
    Transform camera = Camera.main.transform;
    Ray ray = new Ray(camera.position, camera.rotation *
      Vector3.forward );
    RaycastHit hit;
    GameObject hitButton = null;
```

```
        PointerEventData data = new PointerEventData
          (EventSystem.current);
        if (Physics.Raycast (ray, out hit)) {
          if (hit.transform.gameObject.tag == "Button") {
            hitButton = hit.transform.parent.gameObject;
          }
        }
        if (currentButton != hitButton) {
          if (currentButton != null) { // unhighlight
            ExecuteEvents.Execute<IPointerExitHandler> (currentButton,
              data, ExecuteEvents.pointerExitHandler);
          }
          currentButton = hitButton;
          if (currentButton != null) { // highlight
            ExecuteEvents.Execute<IPointerEnterHandler>
              (currentButton, data,
              ExecuteEvents.pointerEnterHandler);
            countDown = timeToSelect;
          }
        }
        if (currentButton != null) {
          countDown -= Time.deltaTime;
          if (clicker.clicked() || countDown < 0.0f) {
            ExecuteEvents.Execute<IPointerClickHandler>
              (currentButton, data,
              ExecuteEvents.pointerClickHandler);
            countDown = timeToSelect;
          }
        }
      }
    }
```

当一个按钮被高亮时，countDown 计时器被开启。当它为 0 时，我们认为是一次点击，你运行起来了吗？不错吧！

所以，这是一个相对复杂的项目，目标是创建一个带有按钮的仪表板，可以打开和关闭管子。我们把它分解成几步，每次按步骤添加对象和组件，然后测试每一步的结果，确保在下一步之前运行正确。如果你想尝试不测试就一次性跑通，事情（大概率）可能会变糟糕，而且会变得更难找出问题卡在哪里。

额外挑战：这个功能可以被进一步地加强以满足不同的需求。比如，它可以被用来在倒计时器运行时给用户一个反馈，可以通过选择光标的动画效果（比如同心圆）显示。另外，当点击事件执行时给出进一步的反馈。例如，**Button UI** 对象

有一个 **Transition** 选项可以调用 **Animation**，也许会有用处。另外，考虑一下语音提示。

5.8　带有头部姿势的响应式 UI

我将讨论的最后一项 UI 技术，它的 UI 元素不需要一直可见，而是根据游戏控制的情况被调用，我称之为*响应式 UI*。

举个例子，在传统的视频游戏中，你可能有一个一直可见的弹药仪表盘。在虚拟现实中，你可以等待用户向下看手中的武器，然后武器上的弹药刻度会发光显示其状态。

在本例中，我们要让刚才创建的那个仪表板只在我们推断出用户有访问它的意图时才出现。这个机制应该感觉起来很自然，你迅速地向下看你的脚，然后仪表板会滑出去，而当你移开视线几秒后仪表板会再次缩回去。

5.8.1　使用头部的位置

让我们试着只用摄像机角度判断你是否正向下看，比如 60° 以内向下看。在 GameController 上创建一个命名为 HeadGesture.cs 的新脚本，检测你是否脸朝向下方，代码如下：

```
using UnityEngine;
using System.Collections;

public class HeadGesture : MonoBehaviour {
  public bool isFacingDown = false;

  void Update () {
    isFacingDown = DetectFacingDown ();
  }

  private bool DetectFacingDown () {
    return (CameraAngleFromGround () < 60.0f);
  }

  private float CameraAngleFromGround () {
    return Vector3.Angle (Vector3.down, Camera.main.transform.rotation
* Vector3.forward);
  }
}
```

此脚本定义了一个 HeadGesture 类，其中有一个很有用的公有变量，isFacingDown，在每一次 Update() 中更新。我把它定义成类变量让它可以被重用，并且其他脚本也可以访问它。细节都被分解成了更小的单一目的的函数。

DetectFacingDown() 函数检测摄像机的角度是否是以 60° 以内朝下。

在 CameraAngleFromGround() 函数中，我们得到当前摄像机相对于直下方的角度，返回一个 0 ～ 180 的角度值。

本质上，Unity 使用一个 **Quaternion** 数据结构表达三维的朝向和旋转，对于运算和准确性来说都是最佳的。Unity 编辑器让我们指定角度，像欧拉（Euler，发音同 oiler）旋转是绕着 *x*, *y*, *z* 轴旋转的角度。参见 http://docs.unity3d.com/ScriptReference/Quaternion.html。

现在，同样在 GameController 上创建另一个新的脚本，命名为 FlippinDashboard.cs，当你向下看时翻开仪表板，代码如下：

```
using UnityEngine;
using System.Collections;

public class FlippinDashboard : MonoBehaviour {
  private HeadGesture gesture;
  private GameObject dashboard;
  private bool isOpen = true;
  private Vector3 startRotation;

  void Start () {
    gesture = GetComponent<HeadGesture> ();
    dashboard = GameObject.Find ("Dashboard");
    startRotation = dashboard.transform.eulerAngles;
    CloseDashboard ();
  }

  void Update () {
    if (gesture.isFacingDown) {
      OpenDashboard ();
    } else {
      CloseDashboard ();
    }
  }

  private void CloseDashboard() {
```

```
    if (isOpen) {
      dashboard.transform.eulerAngles = new Vector3 (180.0f,
        startRotation.y, startRotation.z);
      isOpen = false;
    }
  }

  private void OpenDashboard() {
    if (!isOpen) {
      dashboard.transform.eulerAngles = startRotation;
      isOpen = true;
    }
  }
}
```

此脚本引用了其他的 HeadGesture 组件。事实上，它是从同一个 GameController 中找到这个组件实例的，然后在每次更新时判断用户是否下在向下看（`gesture.isFacingDown`）。在这个例子中，我们使用了另一个脚本中的 `HeadGesture` 类的公有变量，而不是用 Unity 编辑器的 **`Inspector`** 面板中的那一个。

在 `Start()` 函数中，我们把仪表板的 `startRotation` 初始化在打开的位置，就像在 Unity 编辑器中设置的那样。然后，我们关闭仪表板。

`Update()` 函数检测用户是否处于 `isFacingDown` 的姿势，然后打开仪表板。另外，函数关闭仪表板。`CloseDashboard()` 函数通过把 *X* 旋转设置到 180° 以关闭，但只在仪表板已经打开时才这么做。`OpenDashboard()` 函数恢复旋转值到打开状态，而只在仪表板关闭时才这么做。

当你 *Play* 场景时，仪表板开始是折叠的；当你向下看时，它会打开；当你视线移开时，它会再次折叠起来。这就是响应式 UI！

5.8.2　使用头部的姿势

有很多改进行这类行为的途径。有一个用于替代通过简单地向下看打开仪表板的想法是，让用户必须迅速向下看，就好像向下看的姿势中有这个意图。换句话说，如果你偶然地向下看，仪表板不会打开；如果你迅速地向下看，仪表板才会打开。这是一个简单的头部姿势输入的例子。

在程序中把功能单元分离到类的自身中。比如，我们在 HeadGesture 中所做的，让代码更模块化、可重用、可测试、可维护。

让我们像下面这样修改 HeadGesture 脚本：

```
using UnityEngine;
using System.Collections;

public class HeadGesture : MonoBehaviour {
  public bool isFacingDown = false;
  public bool isMovingDown = false;

  private float sweepRate = 100.0f;
  private float previousCameraAngle;

  void Start () {
    previousCameraAngle = CameraAngleFromGround ();
  }

  void Update () {
    isFacingDown = DetectFacingDown ();
    isMovingDown = DetectMovingDown ();
  }

  private float CameraAngleFromGround () {
    return Vector3.Angle (Vector3.down, camera.transform.rotation
      * Vector3.forward);
  }

  private bool DetectFacingDown () {
    return (CameraAngleFromGround () < 60.0f);
  }

  private bool DetectMovingDown () {
    float angle = CameraAngleFromGround ();
    float deltaAngle = previousCameraAngle - angle;
    float rate = deltaAngle / Time.deltaTime;
    previousCameraAngle = angle;
    return (rate >= sweepRate);
  }
}
```

我们刚才检测了向下的位移并设置了一个公有的 isMovingDown 函数。

DectMovingDown() 函数在每次的 Update() 被调用时取得摄像机的 *X* 旋转值（角度），并与上一帧的 previousCameraAngle 比较。然后，我们计算旋转率（rate），再检测 rate 是否超过临界值（sweepRate）。这算是一次有效姿势。我已经发现了以 100.0 的 sweepRate 运行非常好，你可以试试看。

把 isMovingDwon 检测添加到 FlippingDashboard 脚本中，并且替换 Update()，如下：

```
void Update() {
  if (gesture.isMovingDown) {
    OpenDashboard ();
  } else if (!gesture.isFacingDown) {
    CloseDashboard ();
  }
}
```

我发现它有一点儿太敏感了，在你再次向上看时，在关闭仪表板之前添加一个 2s 的延迟会有帮助。添加一个计时器，如下：

```
private Vector3 startRotation;
private float timer = 0.0f;
private float timerReset = 2.0f;

...

void Update() {
  if (gesture.isMovingDown) {
    OpenDashboard ();
  } else if (!gesture.isFacingDown) {
    timer -= Time.deltaTime;
    if (timer <= 0.0f) {
      CloseDashboard ();
    }
  } else {
    timer = timerReset;
  }
}
...
```

总结一下，我们有一个在开始时处于折叠位置而当你向下看则会打开的仪表板。首先，我们简单地使用你的注视角来决定是否打开它。然后，我们对它进行加强，让它只有在你以向下看的姿势（似乎是表明意图）时才打开，而偶然地向下看则不会打开。

额外挑战：你能说明为如何添加一个 side-to-side 的头像姿势吗？上下点头是打开，左右摇头是关闭。点击你的头盔的侧边如何呢？当作是一次点击？这个主意得到了评论人员们的诸多好评和差评（http://www.reddit.com/r/oculus/comments/2cl3wp/tap_the_side_of_the_rift_to_select/）。

当实现头部姿势输入时，重要的是区分偶然向四周看和一个有意图的姿势，以及维护用户在虚拟现实体验中的沉浸感。

我们当前的实现是一种小伎俩。比如，我们只向后看一帧（如第 1/60s），然而如果计算更多帧则姿势会更好。另外，姿势肯定会更复杂，点头甚至可能会有加速度、减速度和弹力。此外，人与人之间的差异也很大。有一些新兴的第三方承诺提供更完整且更健康的解决方案，其中包括像手部和身体感应器的姿势检测的所有头部姿势。

当实现一个头部姿势输入时，重要的是代码可以区分偶然向四周看和有意图的姿势，并维护虚拟现实体验中的沉浸感。

我们需要很多实验来获得对于在虚拟现实中如何使用姿势的更好理解。我鼓励你继续探索这个新的用户接口词汇。

小结

在 Unity 中，UI 是基于一个 canvas 对象的，而事件系统包括按钮、文本、图片、滑块和输入框，可以被组装和连接到场景的对象上。

本章中，我们近距离地查看了各种世界坐标系中的 UI 技术，以及它们在虚拟现实项目中如何被使用。我们考虑虚拟现实中的 UI 与传统视频游戏中和桌面应用的 UI 的不同。另外，我们实现了其中的好几个，演示了它们是如何在你的项目中被构造、编码和使用的。我们的 C# 脚本变得稍微有一点点进阶，深入探索 Unity 引擎的 API 和模块化编程技术。

你现在有一个更大的资料库可以去实现你在 VR 项目中制作 UI。本章中的有些例子可以直接应用到你的项目中，但是，不是所有代码都需要自己实现。VR UI 工具在 VR 头盔的 SDK 中被越来越多地提供，以及开源虚拟现实中间件项目和第三方 Unity 资源商店包。

下一章中，我们将添加一个第一人称角色控制器到我们的场景中。我们将学习虚拟角色和在 VR 中控制导航的方式，让我们舒服地在虚拟世界中向四周移动。另外，我们将学习如何管理虚拟现实体验的一个负面——VR 晕动症。

第 6 章 Chapter 6

第一人称角色

Ninja on Segway，Alaric Holloway 的插图，已授权

太奇怪了，我们已经在这本关于 VR 的书里走得够远了，但是仍然还在使用一个固定的第三人称摄像机！这是有意为之。

当一个人开始构建一个 VR 应用的时候，典型的方法是立即把用户作为第一人称角色直接放进场景中。毕竟，戴着 VR 头盔天生就是第一人称视角。然而，虚拟现实也并不总是第一人称视角的。从第三人称视角观察和控制行为（例如情景剧中）也是一个合情合理的方式。实际上，一些研究表明一些快节奏的动作游戏中，如果使用第一视角进行 VR 游戏会不可避免地引起晕动症，使用第三人称视角来代替就会更好。

这就是说，本章中，我们将会把自己放进一个可控制的第一人称角色中，并且探索在虚拟世界中移动的技术。

本章中，我们将讨论以下话题：

- Unity 的角色对象和组件。
- 使用按钮和 / 或者头部移动控制导航。
- 校正你的虚拟角色（替身）。
- 将你的头部从你的身体里分离出来。
- 探索在 VR 中维持自我意识的技术。
- VR 晕动症的有关问题。

注意，本章中的项目均是相互独立的，不直接需要本书中其他章节的内容。如果你决定跳过它们或者不保存你的成果，也没有问题。

6.1 深入理解 Unity 角色

第一人称角色是 VR 项目中的一个关键点，值得我们彻底理解它的组件。所以，我们在构建一个项目之前，最好详细了解一下 Unity 为我们提供的内置组件和标准资源。

6.1.1 Unity 组件

你可能知道，每个 Unity 游戏对象都包含一系列相关联的 components。Unity 包含很多种内置组件，你可以通过主菜单栏的 **Component** 菜单中查看。每个组件为它从属的对象添加一些属性和行为。一个组件的属性可以通过 Unity 编辑器的 **Inspector** 面板和脚本访问。一个附加到游戏对象的脚本也是一种组件的类型，在 **Inspector** 面板中你同样可以设置它的属性。

可以用来实现第一人称角色的组件类型包括：**Camera**、**Character Controller**、**RigidBody** 和各种各样的脚本。让我们回顾一下这些标准组件。

1. Camera 组件

Camera 组件指定了各种观察参数，这些参数用于在每帧更新的时候渲染场景。拥有 Camera 组件的任何对象都被认为是一个摄像机对象。从一开始我们就已经在场景中使用了一个摄像机，而且我们已经用自己写的脚本来访问了它。

一个立体 VR 摄像机对象渲染两个视区，对应两只眼睛。在 VR 中，摄像机控制器脚本从头盔运动传感器中读取数据用以检测当下的头部姿态（位置、方向和旋转），然

后设置到对应的摄像机变换值。

2. RigidBody 组件

当你添加一个刚体（RigidBody）组件到任意游戏目标上时，它将会受益于**物理引擎**提供的计算。刚体拥有重力、质量、阻力等参数。在游戏运行时，物理引擎会计算每个刚体的动量 (momentum)（质量、速度和方向）。

刚体会与其他刚体发生作用。例如，如果它们之间发生碰撞，它们将会相互弹开，并且可以通过带有摩擦力和弹性因子参数的物理材质（physic material）控制相互作用的参数值。

一些刚体被标记为运动学（kinematic）的，指通常在物体由动画或者脚本驱动时使用。碰撞不会影响运动学物体，但是它们将仍然影响其他刚体的运动。它经常被用在一些关节（joints）串联一起的物体上，例如一些被人型骨骼连接的物体或者一个钟摆。

任何刚体，给它一个摄像机对象都能成为一个刚性的第一人称角色。然后，你可以添加脚本来处理用户，如移动、跳跃、向四周看等输入操作。

3. Character Controller 组件

与 RigidBody 类似，**Character Controller**（简称 CC）组件也用于碰撞检测和角色移动。它同样需要脚本处理用户输入来移动、跳跃、向四周看。然而，它并不自动拥有内置的物理。

CC 组件是专门为角色对象设计的，因为游戏中的角色通常不能真实地表现出像其他基于物理的物体那样的行为。它可以用于替代刚体。

CC 组件拥有一个内置的 Capsule Collider 内为来检测碰撞。然而，它并不能自动使用物理引擎来响应碰撞。

例如，如果一个 CC 对象撞击了一个刚体（如一堵墙），它将会停止，不会被反弹回来。如果一个刚体，例如一个飞行的砖块撞击了一个 CC 对象，砖将会根据其属性获得偏移（反弹），但是 CC 对象不会有任何影响。当然，如果你想要在 CC 对象上包含这样的行为，可以通过在你的脚本中编写代码达到目的。

CC 组件对于它的脚本 API 中的一种作用力——重力提供了一个极好的支持。内置的参数具体来说与保持物体的脚在地面上有关。例如，**Step Offset** 参数定义了角色的一步能迈出多高，从而可以让他跨越挡在路上的障碍物。同样的，**Slope Limit** 参数表示一个坡有多陡，从而判断它是否可以被认为是一堵墙。在你的脚本中，你可以使用 Move( )

方法和 IsGrounded 变量来实现角色的行为。

除非你使用脚本编写它，CC 对象并没有动量并且会突然停止。它非常精确，但是这同样导致了一些不稳定的位移。相反对于 Rigidbody 对象，会感觉很流畅因为它们拥有动量，加速度和减速度，并且符合物理规律。在 VR 中，我们喜欢两者的结合。

6.1.2 Unity 的 Standard Assets

在 Unity 中的 **Standard Assets** 的 **Character** 包中带有一系列第三人称和第一人称预制件对象。这些预制件对象的对比在下面的表格中。

Prefab	Components
ThirdPersonController EthanBody EthanGlasses EthanSkeleton	ThirdPersonController Transform Animator Rigidbody Capsule Collider Third Person User Control (Script) Third Person Character (Script)
AIThirdPersonController EthanBody EthanGlasses EthanSkeleton	AIThirdPersonController Transform Animator Rigidbody Capsule Collider Nav Mesh Agent AI Character Control (Script) Third Person Character (Script)

Prefab	Components
FPSController FirstPersonCharacter	FPSController Transform Character Controller First Person Controller (Script) Rigidbody Audio Source
RigidBodyFPSController MainCamera	RigidBodyFPSController Transform Rigidbody Capsule Collider Rigidbody First Person Controller (Script)
OVRPlayerController ForwardDirection OVRCameraRig	OVRPlayerController Transform Character Controller OVR Gamepad Controller (Script) OVR Player Controller (Script) OVR Main Menu (Script)

让我们详细讨论一下这个些预制件。

1. ThirdPersonController

我们在第 2 章和第 4 章中已经分别使用了这两个第三人称预制件：ThirdPersonController 和 AIThirdPersonController。

ThirdPersonController 预制件拥有一个子对象，定义了角色的身体，也就是我们的朋友 Ethan。他是一个装配而成的虚拟角色（在 .fbx 文件中）；这意味着人型动画可以应用到其的身上，使他走动、跑步、跳跃等。

ThirdPersonController 预制件使用一个 Rigidbody，它包含了用于检测碰撞的物理和胶囊碰撞器。

ThirdPersonController 包含两个脚本。ThirdPersonUserControl 脚本获取用户的输入，例如按下键盘，然后通知角色移动、跳跃等。ThirdPersonCharacter 脚本实现了移动背后的物理，称之为动画，在跑动和蜷缩等时候需要它。

2. AIThirdPersonController

AIThirdPersonController 预制件和 ThirdPersonController 预制件一样，但是前者多了一个 NavMeshAgent 和一个 AICharacterControl 脚本，约束了角色可以在场景中的哪里以及如何移动。如果你回想一下，在第 4 章中我们使用 AICharacterController 让 Ethan 在场景中四处游荡并且避免碰到物体。

3. FirstPersonController

FPSController 预制件是一个第一人称控制器，CC 组件和 RigidBody 都有使用。它还附加了一个摄像机子对象。当角色移动时，摄像机会跟随它移动。

第三人称和第一人称预制件的关键区别在于子对象。第三人称控制器预制件拥有一个装配的人形子对象，而第一人称控制器预制件拥有的是一个摄像机子对象。

它的身体质量被设置成了一个非常小的值（1），并且 IsKinematic 呈开启状态。这意味着它将拥有有限的动量且不会作用到其他刚体，但是它可以由动画驱动。

它的 FirstPersonController 脚本提供了大量的参数用于跑步、跳跃、脚步声以及更多。此脚本同样包含了用于点头（head bob）的参数和动画，它在角色移动的过程中以自然的方式反弹摄像机。如果你在你的 VR 项目中使用 FPSController 脚本，一定要保证禁用任何点头（head bob）功能，不然你可能会吐在你的键盘上！

4. RigidBodyFPSController

RigidBodyFPSController 预制件是拥有一个 RigidBody 组件但没有 CC 组件的第一人称控制器。像 FPSController 一样，它也有一个摄像机子对象。当角色移动时，摄像机随之移动。

一个 RigidBodyFPSController 预制件的身体质量更可观，被设置为 10，并且不是动力学的。也就是说，它在跟其他物体碰撞的时候会被弹开。它有一个单独的带 ZeroFriction 物理材质的物理碰撞组件。RigidBodyFirstPersonController 脚本不同于 FPSController，但是前者有很多相似的参数。

为什么我会讲得这么详细？如果你用 Unity 构建过任何非 VR 项目，那么你肯定已经使用过这些预制件了。然而，你可能没有太关注过它们是如何被组装起来的。虚拟现实是第一人称角度的体验。我们的实现工具是 Unity。真正地了解 Unity 的工具从而管理和控制第一人称体验是至关重要的。

6.2 制作第一人称角色

为了制作第一人称功能，让我们使用敏捷（agile）方式进行开发。这意味着（在某种程度上）我们由开始时根据一些需求定义我们的新功能，或者说故事。然后，我们以增量的方式构建和测试这些功能，一次一个需求，随着我们的项目进行不断迭代和打磨。验证尝试不单是允许，而是我们所鼓励的。

敏捷软件开发是描述方法论的一个广义术语，它鼓励以一种容易响应变化打磨需求的方式进行小型的增量和迭代开发。

功能：对于第一人称角色，当我开始走动，我将以我看的方向穿过场景，直到我决定要停下来。

以下是这个功能的需求：

- ❑ 在你直视的方向上移动。
- ❑ 保持脚在地面上。
- ❑ 不要穿过固体。
- ❑ 不要从世界边缘掉下去。
- ❑ 跨越小型物体并且处理崎岖路面。

❑ 通过头部姿势（向下看）或者点击输入设备来开始或者停止移动。

这听起来是合理的。

6.2.1　在直视的方向上移动

我们已经有一个 MeMyselfEye 对象包含了摄像机装置。我们要把它变成一个第一人称控制器。我们的第一个需求是沿着你直视的方向上移动。添加一个名为 HeadLookWalk 的脚本。为了保持简单性，让我们开始执行下面的步骤：

1. 在 **Hierarchy** 面板中，选中 MeMyselfEye 对象。

2. 在 **Inspector** 面板中，选择 **Add Component | New Script**，并且命名为 HeadLookWalk。

然后，打开脚本并且输入以下代码：

```
using UnityEngine;
using System.Collections;

public class HeadLookWalk : MonoBehaviour {
  public float velocity = 0.7f;

  void Update () {
    Vector3 moveDirection = Camera.main.transform.forward;
    moveDirection *= velocity * Time.deltaTime;
    transform.position += moveDirection;
  }
}
```

人类的正常步速大约为 1.4m/s，让我们把速度减半为 0.7m/s。在 `Update()` 方法中，我们检查了摄像机面对的方向（`camera.transform.forward`）并且以当前速度在这个方向上移动 MeMyselfEye 的变换位置。

注意用于变量的自修改的快捷编码方式。最后两行代码本来可以这样写：

```
moveDirection = moveDirection * velocity * Time.deltaTime;
transform.position = transform.position  + moveDirection;
```

这里，我使用 *= 和 += 操作符替代它们。

保存脚本和场景并且在 VR 中尝试。

当你向前看的时候，你会向前移动。向左看，就会向左移动。向右看，就会向右移动。它工作得很好！

向上看……哇！！你能想到这一点么？我们正在飞翔！你可以上、下，向各个方向移动就像是超人或者一个无人机的驾驶员一样。现在，MeMyselefEye 没有质量和物理，并且不响应重力。尽管如此，它还是满足了我们的需求——在你直视的方向上移动。那么，让我们继续吧。

6.2.2 保持脚着地

下一个需求是让你的脚保持在地面上。我们知道 GroundPlane 是平面并且位于 $Y = 0$ 的位置。那么，我们把这个简单的约束条件添加到 HeadLookWalk 脚本中：

```
void Update () {
  Vector3 moveDirection = Camera.main.transform.forward;
  moveDirection *= velocity * Time.deltaTime;
  moveDirection.y = 0.0f;
  transform.position += moveDirection;
}
```

保存脚本并在 VR 中试试。

还不错。现在，我们可以在 $Y = 0$ 的平面上移动了。

另一方面，你可以像个幽灵，轻易地穿透立方体、球体和其他物体。

6.2.3 不要穿透固体

第三个需求——不要穿透固体。这里有个方案。给它一个 Rigidbody 组件并且让物理引擎起作用，可以通过执行以下步骤获得：

1. 在 **Hierarchy** 面板中，选中 MeMyselfEye 对象。

2. 在 **Inspector** 面板中，点击 **Add Component | Physics | Rigidbody**。

在 VR 中尝试。

哇！这是什么？它只有不到 1s，但是当你撞上立方体，你便不受控制了，就像在电影《重力》中的太空漫步出错一样。好吧，这是为你准备的 Rigidbody。力作用在各个方向上。让我们添加一些约束，如下：

在 **Inspector** 的 **Rigidbody** 面板中，取消选中 **Freeze Position : Y** 和 **Freeze Rotation : X** 和 **Z**。

在 VR 中尝试。

现在效果很好了！你可以在直视的方向上移动，你并没有在飞行（Y 轴位置被约束），而且你也不能穿透固体。你可以滑过它们因为只有 Y 旋转被允许。

假设你的 KillTarget 脚本仍然在运行（第 4 章中），你可以一直注视 Ethan 直到他爆炸。就这么做吧，让 Ethan 爆炸……哇！我们被爆炸冲击出来，并且再次失控地旋转。也许我们还没有为这个强大的物理引擎准备好。我们应该可以在脚本中找到，但我们暂时先丢掉 Rigidbody 这个想法。我们将会在下章继续讨论它。

你可能回想起来 CC 包含一个胶囊碰撞器并且支持受碰撞约束的移动。我们将尝试用它替代，如下：

1. 在 **Inspector** 面板中，点击 **Rigidbody** 面板的齿轮图标并且选择 **Remove Component**。
2. 在 **Inspector** 面板中，点击到 **Add Component | Physics | Character Controller**。

修改 HeadLookWalk 脚本，如下：

```
using UnityEngine;
using System.Collections;

public class HeadLookWalk : MonoBehaviour {
  public float velocity = 0.7f;

  private CharacterController controller;

  void Start () {
    controller = GetComponent<CharacterController>();
  }

  void Update () {
    Vector3 moveDirection = Camera.main.transform.forward;
    moveDirection *= velocity * Time.deltaTime;
    moveDirection.y = 0.0f;
    controller.Move(moveDirection);
  }
}
```

不直接更新 transform.position 了，我们调用内置的 `CharacterController.Move()` 函数来满足我们的目的。它知道角色的行为有某些约束。

保存脚本并且在 VR 中尝试。

这一次，当我们碰到物体（立方体或者球体）的时候，我们差不多可以越过它然后悬浮于空中。`Move()` 函数没有为我们把重力应用到场景中。我们需要把它添加到脚本中，这并不难（请查看 Unity API 文档，http://docs.unity3d.com/ScriptReference/CharacterController.Move.html）。

然而，还有一个更简单的方法。`CharacterController.SimpleMove()` 函数为我们在移动时应用了重力。只要用下面的一行代码替换整个 `Update()` 方法即可：

```
void Update () {
  controller.SimpleMove(Camera.main.transform.forward *
    velocity);
}
```

SimpleMove() 函数处理重力并且处理 Time.deltaTime。所以，我们要做的就是提供移动方向矢量给它。同样的，由于它引入了重力，我们也不需要 $Y = 0$ 的约束了。这样就更简单了。保存脚本并且在 VR 中尝试。

太棒了！我想我们快要完成所有的需求了。不要走出边缘……

6.2.4 不要在边缘坠落

现在我们拥有重力了，当我们走出地平面边缘的时候，你会陷入被遗忘的角落。修复这个问题并不是第一人称角色的事。只需要在场景中添加一些栏杆。

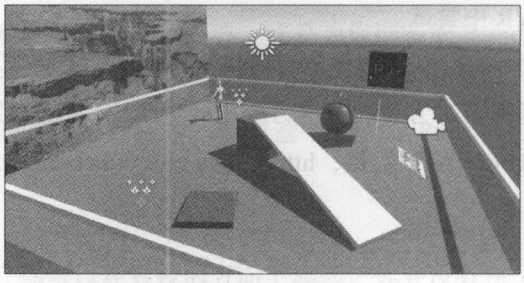
使用立方体，把它们缩放到希望的厚度和长度，然后移动它们到指定位置。自己做吧。我不会一步步地给你介绍如何做。例如，我使用了这些变换值：

- **Scale**：0.1, 0.1, 10.0
- 栏杆 1，**Position**：−5，1，0
- 栏杆 2，**Position**：5，1，0
- 栏杆 3，**Position**：0，1，−5；**Rotation**：0，90，0
- 栏杆 4，**Position**：0，1，5；**Rotation**：0，90，0

在 VR 中尝试。试着穿越栏杆。哈！现在这下安全了。

6.2.5 跨越小物体并处理崎岖路面

在我们这样做的时候，添加一些物体在上面用于行走和跨越，例如一个斜坡和其他障碍物。结果看起来像下面这样：



在 VR 中尝试。走上斜坡并且走下立方体。嘿，有趣！

CC 组件完成了跨越小物体和处理崎岖路面的需求。你可能需要通过它的 **Slope Limit** 和 **Step Offset** 设置来调整它。

6.2.6　开始和停止移动

下一个需求是使用点击或者头部姿势来开始和停止移动。首先将通过我们在第 3 章中写过的 Clicker 类来实现点击，这个类检查是否按下键盘、鼠标或者 Cardboard 触发器的按键。

修改 HeadLookWalk 脚本，如下所示：

```
using UnityEngine;
using System.Collections;

public class HeadLookWalk : MonoBehaviour {
  public float velocity = 0.7f;
  public bool isWalking = false;

  private CharacterController controller;
  private Clicker clicker = new Clicker();

  void Start() {
    controller = GetComponent<CharacterController> ();
  }

  void Update () {
    if (clicker.clicked()) {
      isWalking = !isWalking;
    }
    if (isWalking) {
      controller.SimpleMove (Camera.main.transform.forward *
        velocity);
    }
  }
}
```

通过添加一个 `isWalking` 标志位，我们可以切换前进的开和关，可以通过按下一个键来发信号。

6.2.7　使用头部姿势开和关

要添加用于行走的一个头部姿势触发器，我们可以使用我们在第 5 章中创建的

HeadGesture 类（假设它们还是附加到了你的场景中的 GameController 对象上）。我们可以修改 HeadLookWalk 脚本，但是让我们把它们放在另一个单独的脚本中，并且把它们绑在一起，如下：

1. 在 **Hierarchy** 面板中，选中 MeMyselfEye。
2. 在 **Inspector** 面板中，选择 **Add Component | New Script**，并且命名为 GestureWalk。

然后，打开脚本并且编写：

```
using UnityEngine;
using System.Collections;

public class GestureWalk : MonoBehaviour {
  private HeadLookWalk lookWalk;
  private HeadGesture gesture;

  void Start () {
    lookWalk = GetComponent<HeadLookWalk> ();
    gesture = GameObject.Find ("GameController").
GetComponent<HeadGesture> ();
  }

  void Update () {
    if (gesture.isMovingDown) {
      lookWalk.isWalking = !lookWalk.isWalking;
    }
  }
}
```

我们通过从“注视—行走”控制器中分离头部姿势输入使得我们的代码更加模块化，并修改了 HeadLookWalk 类的公有变量 isWalking。

现在我们同样可以通过点头的操作来切换前进的开关。看到没，不用手！你可以在 VR 中开始和停止四处走动，而不需要键盘、鼠标或者游戏控制器。（应该指出这个例子是为了好玩。我并不提倡在你的应用中用这个不错的机制。）

6.3 用户校准

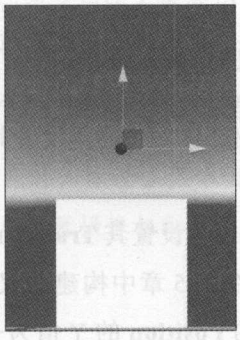
你有多高？你现在是站着还是坐着的？我指的是真实生活中的你自己还是在虚拟世界中的你的玩家角色？如果虚拟的你没有校准过，那么你的第一人称 VR 体验可能感觉不好。

注意，我指的不是你的 VR 头盔的物理配置。举个例子，**Oculus Configuration Utility** 让你对**眼距**（eye relief）（HMD 镜头距离你的眼睛有多远）和**瞳距**（Interpupillary Distance，IPD）（双眼间的距离）进行校准，这将影响由 SDK 驱动程序执行的底层变形渲染。然而，那些对于想要一个满意的体验来说很重要。我指的是对在场景中正在游戏的你的玩家角色的校准。

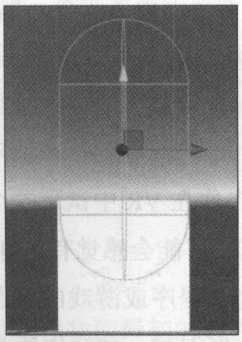
6.3.1　角色的身高

我不知道你是否已经留意过第一人称摄像机的高度，但是看起来我们已经大变样了。本章之前，摄像机的视线高度被设置到了与 Ethan 四目相对的高度。现在，我们看到的是它的胸部高度，发生了什么？

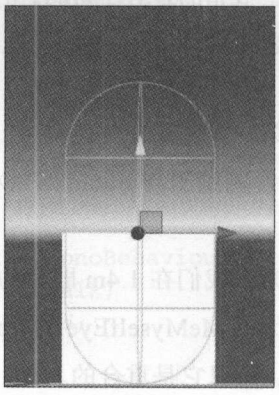
如果你一路顺着第 3 章走过来，我们在 1.4m 即大约 Ethan 的视线高度处创建了一个 MeMyselfEye 的对象，如下图所示（MeMyselfEye 的 **Position** 组件的 $Y=1.4$）。摄像机子对象的相对位置是（0，0，0），所以它是重合的。



然后，在本章的前面的章节中，我们给 MeMyselfEye 添加了一个角色控制器组件，它包括一个默认高度为 2.0 的 Capsule Collider。Transform.position 预制件现在出现在胶囊的中心，如下图所示：



接着，当我们 play 场景（并调用 SimpleMove）时，重力开始生效，下落这个对象让它接触地面，把中心和摄像机移动到 $Y=1.0$，如下图所示。（当你按下 Play 模式时你会立即发现这个问题）：



我们需要收拾一下。

功能：作为一个第一人称角色，我的视线高度应该在 1.4m。

执行下面的步骤：

1. 在 **Hierarchy** 面板中选择 MeMyselfEye，设置其 **Transform** 组件的 **Position** 值 Y 值为 1.08。

2. 选中其子的 VR **Main Camera**，设置其 **Transform** 组件的 **Position** 的 Y 值为 0.4。

3. 如果你在场景中还有我们在第 5 章中构建的仪表板，选择那个 **Dashboard** 子对象，设置其 **Rect Transform** 组件的 **Position** 的 Y 值为－0.6。

胶囊的完整高度是 2.0。那么，我们设置其位置的 $Y=1.08$。为什么多出 0.08？它有一个默认的 **Skin Width** 值为 0.08（skin 是 Unity 让角色通过狭窄通道的容差系数）。我们想让摄像机的高度在胶囊中心之上 0.4 处。我们还要像之前那样调整仪表板到视线之下 1m。（这里是我的设置，比如你可以设置 skin 为 0.001 而胶囊的高度值为 1.6，以及进行其他相应的调整。）

6.3.2 玩家的真实身高

如果你实际上 1.6m 高，那么当你在 VR 中试玩这个角色时这个视线高度可能感觉差不多。如果你实际上更高（或更矮），可能会感觉有点麻烦。

现在，你需要问问你自己，这个程序或游戏的意图是什么？你正像一个 Ethan 那么高的孩子一样玩耍？作为替代，你想创建一种让你感觉到你被传送进虚拟世界的 VR 体

验吗？你的角色是站立还是坐着的？

第一种情况要简单一些。让我们假设你正在一个 VR 游戏中，让我们假设你是马里奥或路易吉。然后，你正在扮演一个矮个子，而你可以从那个高度看世界。你不会有六尺高，即使你在真实生活中很高。同样，如果你是 Bowser，你需要有大约十尺高，而不管你自己、游戏的设计者有多高都无所谓。

第二种情况，你正在设计一种体验，传送一个真实的用户到一个新的虚拟地点。你会想让摄像机的高度接近真实玩家的视线高度。取决于你的项目，也许选取世界人类平均身高比较好（1.7m）。

如果你想进行微调，你的应用可以有一个校准菜单让用户输入其真实的高度或从列表中选择，比如矮、平均、高。Oculus Rift 包括一个 **Configuration Utility**，让你预定义一个或多个用户配置，包括身高和性别，所有 Rift 应用通过其 API 都可以访问。

如果你的角色是坐着的，视线高度可能不是问题，因为我们习惯了坐在标准高度的桌子旁边（比如约 70cm）而我们的椅子与这个高度相配。

当虚拟现实设备发展到手和身体传感器更普遍的地步时，我们就需要校准工具和用户配置以便帮助玩家感到用它们的虚拟身体时像在家里一样自在。

当你扮演一个游戏中的角色时，你设置摄像机到角色的高度。当你虚拟地传送用户到新的虚拟地点时，你要设置摄像机到他们真实眼睛的高度。

6.3.3　回到中心位置

有时候在 VR 中，头盔中出现的视野不是那么能与你身体的朝向同步。设备的 SDK 提供了函数以便重置头盔关于真实空间的朝向。这通常指的是视野的重定位（**recentering**）。

对于 Oculus Rift，你可以添加一个脚本用于监听特定的按键，比如 *R* 键重定位视图或一个有用的 HUD 菜单按钮并调用 InputTracking.Recenter( )。

Google Cardboard 有一个类似的函数用于重置 HMD 的朝向。当然，它没有位置跟踪功能。通过调用 Cardboard.SDK.Recenter( )，只有朝向和运动传感器被重置。

6.4　保持自我感

在 VR 中，你可以在虚拟空间（像我们最初的透视图摄像机）中只作为一个浮动的一

维眼球、一个标准人类，或者也许某种魁梧的空间生物。我们已经集中在场景中四处走动并讨论了如何把摄像机高度设置为视线高度。这样的结果可能是某些人体验 VR 时能感觉到自己的身体。然而，摄像机的高度实际上只有当在你附近还有一块地平面或一些其他的固定参考点时才有意义。当你像鸟儿、飞机或超人一样飞来飞去时，那么身高可能就不重要了。

在当今物理世界中，我们所有人都真的有身体，以及我们的大脑的一种期望。围绕着自我感知的问题可能很大程度上是心理学、哲学的问题，甚至是宗教问题。另外，令人难以想象的是，就这一点而言，VR 也许终于能够做到。对于现在，我们应该专注于我们的需求，以便让我们的 VR 体验能让我们的顾客感到舒适。

6.4.1 身首分离

当仔细研究虚拟现实头盔时，提到很多的是关于位置跟踪和头部姿势。这对于提供沉浸式 VR 体验来说很重要。不幸的是，你也许只是一个普通人。

我们中的大多数人在我们的 VR 配置中并没有身体跟踪功能，而只有头部跟踪功能（除非你阅读本书的时间远远晚于写书之时，否则你就是一个非常特别的身体跟踪设备的早期采用者，或者你喜欢摆弄 **Microsoft Kinect**）。这意味着当你向四周看（比如从左到右）时，你的软件不知道你是否只移动了你的头部（从脖子以上）或保持你的头部静止而移动你的整个身体（比如你只是转动了你的椅子）。当然，真实生活的身体，像装配出来的虚拟角色一样，也可以扭动肩膀、臀部、双腿等。此外，眼部跟踪是另一件事——不移动你的头部和身体像四周看。

这意味着什么？当戴着一款 VR 头盔时，让我们假设你的第一人称角色有一个虚拟角色身体，假设你在一个其他用户可以看到你的多用户虚拟世界中。如果你转动你的头部，不应该只是你的虚拟角色的头部移动吧？这可以说得通。直到你开始向前行走，你的整个身体可能应该转向面对你行走的方向。

考虑另一个场景。假设你正坐在飞船的驾驶座舱中，你可以移动头部向你座位的四周看。然而，你的输入控制将指挥飞船的走向。

你正在取得这张图片。头部动作控制不需要连接到你的身体（或飞船）控制。但是，我并没有身体跟踪。另外，Rift（和类似的设备）有动作跟踪，而移动 VR 头盔，比如 GearVR 和 Google Cardboard 则没有。

让我们看一下我们能做点什么。

6.4.2　头部和身体

功能：作为一个第一人称角色，当我正在行走时，我可以向下看并看见我的脚朝向我前进的方向；当我没有行走时，而我向四周看，我的脚并没有旋转。

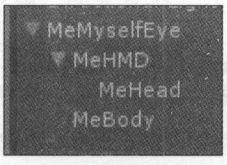
要实现这个功能，我们要把 MeMyselfEye 分成可以独立旋转的头部和身体对象，并给身体加上两只脚，让脚尖指向我们行走的方向，步骤如下：

1. 在 Hierarchy 中选择 MeMyselfEye，点击右键并选择 Create Empty 创建一个空对象，命名为 MeBody。

2. 在 MeMyselfEye 下找到 Main Camera，点击右键并选择 Create Empty 创建一个空对象，命名为 MeHead。

现在，我们的脚本能够知道头部转动的观察方向，它与身体的朝向不同。

然而 Unity 想让你以场景中的一个摄像机的角度思考，我更倾向于以玩家的角度思考。记住玩家戴着一个 HMD。她的头部在 HMD 之中，这个 HMD 也被称为摄像机对象，带有位置跟踪。你不需要重命名这个摄像机，但是如果你这么做了，层次看起来就是这样的：



虽然我们命名我们身体部体的容器为 MeBody，可能会比虚拟的血肉之躯更多。如果你正在开车，它可能包括车身。它可能包括任何附着于你的物体或任何可能携带的物体，比如武器。如果你的应用要支持手持的动作控制器和其他跟踪设备，这些身体总件和脚本也会成为 MeBody 的子对象。

6.4.3　双脚

让我们给身体加上双脚。我已经在本书中包含了一个图片文件：flip-flops.png。（否则，使用任何指示前进方向的东西）。执行下面的步骤：

1. 通过菜单 **Assets** 下的 **Import New Asset**... 导入 flip-flops.png 纹理。

2. 在 **Project** 面板中创建一个新材质，命名为 FlipFlops。

3. 把 flip-flops 纹理拖到 FlipFlops 材质的 **Albedo** 地图上并选择 **Rendering Mode** 为 ***Cutout***。

4. 创建一个 **Quad** 对象（通过菜单 **GameObject | 3D Object | Quad**）作为 MeMyself-Eye/MeBody 的子对象，命名为 Feet，并设置其 **Transform** 组件的 **Position** 为（0，-1，0）而 **Rotation** 为（90，0，0），让它平放在地面上。

5. 在 **Hierachy** 中选择 Feet，把 FlipFlops 材质拖到 **Inspector** 面板上。

如果你现在在 VR 中试运行，双脚在那儿，但是它们没有指向我们行走的方向。我们需要另一个脚本。步骤如下：

1. 在 **Hierarchy** 面板中，选择 MeMyselfEye 对象。

2. 在 **Inspector** 面板中，选择 **Add Component | New Script** 并命名为 BodyWalk。

然后，打开脚本并进行编辑，如下：

```
using UnityEngine;
using System.Collections;

public class BodyWalk : MonoBehaviour {
  private HeadLookWalk lookWalk;
  private Transform head;
  private Transform body;

  void Start () {
    lookWalk = GetComponent<HeadLookWalk> ();
    head = Camera.main.transform;
    body = transform.Find ("MeBody");
  }

  void Update () {
    if (lookWalk.isWalking) {
      body.transform.rotation = Quaternion.Euler (new Vector3
        (0.0f, head.transform.eulerAngles.y, 0.0f));
    }
  }
}
```

在这段脚本中，我们假设了一个特定的层级——脚本是 MeMyselfEye 的一个组件，并且它有一个名为 MeBody 的直属子对象。

现在，当行走时，我们不仅在我们看着的方向上移动 MeMyselfEye，而且在相同方向旋转其身体。因为我们正竖直站立着，身体只关于 y 轴旋转而 X 和 Z 旋转为零。

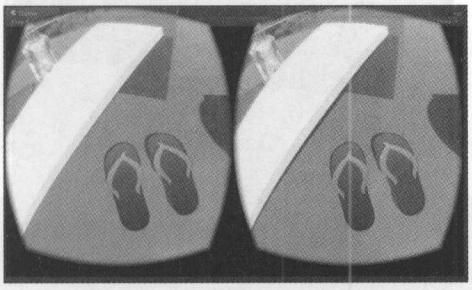
保存脚本，在 VR 中试运行。

你的脚太大了，我的天呐！好吧，这不是一个身体并且脚太大了。另外，它们

在你行走时没有动画，但是我们有些东西需要构建。只展示附着在第一人称角色上的 flipflops，走了一长段让你有着地感觉的路。它可能用一个滑板或者 Segway 代替，你不再是一个浮动的一维眼球点。

如果你在场景中还有第 5 章中的仪表板，你应该移动它并把它作为 MeBody 的子对象，让它跟随你当前行走的方向。

下面的截图展示了这个第一人称角色视图的 flipflops：



6.4.4　身体的虚拟角色

让我们尝试一个完整的虚拟角色。除非你有一个更喜欢的不同的人型，否则我们可以用一个 Ethan 的复本作为我们的身体，步骤如下：

1. 在 **Project** 面板中，在 Standard Assets/ThirdPersonCharacter/Models/Ethan 中找到 Ethan 预制件并把它拖进 **Hierarchy** 面板中的 MeMyselfEye/MeBody 之下。

2. 设置其 **Transform** 组件的 **Position** 为（0，-1，-0.2）。

$Y = -1$，所以，它的脚在地面上。摄像机的 $Z = 0$，所以，$Z = -0.2$ 让 Ethan 的脸在摄像机之后。

在 VR 中试运行。

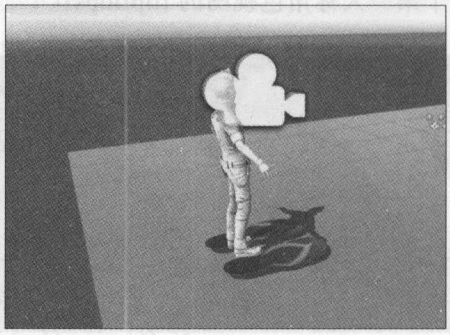
这是一个好的开始。我们可以带着一个身体四处行走。但是，它有些问题，而这也是我们要在这里讨论的，但是现在不需要修复：

- **没有动画**：Ethan 在 T-pose 是静止的。你知道 Ethan 是一个装配人型，他可以完成所有类型的行走和跑动。在 MeMyselfEye 中设置它需要花一些工夫。
- **位置跟踪**：不同于 Cardboard，Oculus Right 以及类似设备的优秀位置跟踪功能，意味着给定的头部动作不仅改变旋转，还改变其处于世界空间中的位置。我们的脚本并不负责此功能。如果你保持头部在相同的位置并且只向四周看，看起来还行。但是，如果你开始前倾、后倾或任何倾斜，你就能发现你自己有真实的离体

体验。（比如，后仰一段距离你将在视图中看见 Ethan 的后脑勺。）

- **无头部约束**：除非你的角色是 Linda Blair（驱魔人）或一只猫头鹰，可能转动你的头部自始至终都很费劲。当你没有处于行走模式时而只是向四周看看，如果你在肩膀以上转动你的目光，脚本应该旋转身体对应以保护你的颈部。

另一个大问题——这些 flipflops 对于我的脚太大了！可以在下图中看到：



当筹划一个尊重真实玩家的第一人称虚拟角色时还有另外一些考虑，比如性别和肤色。

Configuration Utility 中默认的 Oculus Rift demo 在一个办公椅中有玩家设置，但是没有身体。当你向下看时，只能看见一把椅子。其他游戏可能会提供一个没有头的身体，限制 VR 头部姿势与虚拟角色的同步需求。

如果你决定暂时不继续这个完整身体的虚拟角色的想法，将它删除即可。

6.4.5 虚拟的 David le 鼻子

制作虚拟角色的身体的其中一个原因是通过给用户一个空间范围和着地感以帮助其减少晕动症。虚拟现实还很年轻，而人们正在探索新的、创新的方式。也许你不需要一整个身体，也许你只需要一只鼻子。

据一位来自普杜大学的研究人员，David Whittinghill 所说，（http://www.purdue.edu/newsroom/releases/2015/Q1/virtual-nose-may-reducesimulator-sickness-in-video-games.html ）添加一个虚拟的鼻子到你的视野中可以帮助减少晕动症，因为我们正揣着实验的心理，让我们试一下吧。

回到章节 5.2 和 5.3，我们把对象作为 VR 摄像机的子对象，以便让它附着在你的脸上。这听上去像是一只鼻子！执行下面的步骤：

1. 导入本书提供的 david-le-nose.png 纹理。

2. 创建一个新的材质，命名为 Nose，把这个 .png 纹理拖到材质上。

3. 在 Nose 材质中，把它的 **Shader** 设置为 **Unit | Transparent Cutout**。

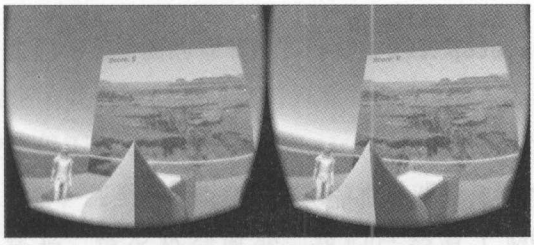
4. 另外，把它的 **Mesh Renderer** 调整为无阴影（反选 **Cast Shadows**、**Receive Shadows**、和 **Use light Probes** 选项）。

5. 找到 **Main Camera** 对象并创建一个新的 quad 让它成为摄像机的子对象，命名为 Nose。

6. 把 **Nose** quad 的 **Transform** 组件的 **Position** 设置为（0，0，0.3），让它在你的脸前面。

7. 把这个 Nose 材质拖到 Nose quad 上。

在 VR 中试运行，喜欢的话就多看一会儿。第一人称角色戴着一只鼻子，展示在下图中。



6.4.6　声音提示

让我们忘记声音是虚拟现实中一个重要的维度。作为另一个快速 demo，让我们添加听得见的脚步到我们的第一人称角色中，步骤如下：

1. 选中 MeMyselfEye，点击菜单 **Add Component | Audio | Audio Source**。

2. 在 **Audio Source** 组件中的 **Audio Clip** 字段中，点击圆形图标（最右边）打开 **Select AudioClip** 对话框并选择 Footstep01。

3. 反选 **Play on Awake** 复选框。

4. 勾选 **Loop** 复选框。

现在，我们可以修改 BodyWalk 脚本，如下：

```
using UnityEngine;
using System.Collections;

public class BodyWalk : MonoBehaviour {
  private HeadLookWalk lookWalk;
  private AudioSource footsteps;
  private Transform head;
```

```
    private Transform body;

    void Start () {
      lookWalk = GetComponent<HeadLookWalk> ();
      footsteps = GetComponent<AudioSource> ();
      head = Camera.main.transform;
      body = transform.Find ("MeBody");
    }

    void Update () {
      if (lookWalk.isWalking) {
        body.transform.rotation = Quaternion.Euler (new Vector3
          (0.0f, head.transform.eulerAngles.y, 0.0f));
        if (!footsteps.isPlaying) {
          footsteps.Play ();
        }
      } else { // not walking
        footsteps.Stop ();
      }
    }
}
```

现在，不管你是否在行走，你都能听到脚步声。

6.5 移动、传送和传感器

我们只实现了一个简单的基于注视的机制用于在虚拟现实场景中移动。你当然也可以决定继续用键盘、手柄或一个游戏板像传统视频游戏一样控制你的角色。用于移动和传送控制的新技术下在不断地被尝试。这里有一些想法：

- **通过注视行走**：按照你注视的方向行走。你的脚站在地面上移动。这是我们之前实现的机制。
- **悬空飞盘**：走到一个悬空飞碟上开始移动，当移动完成时走下来。这需要一种方式标识走上去和走下来的动作。
- **平衡车**：与悬空飞碟类似，但是你通过前倾身体向前移动和转身，通过后倾身体来减速和停止移动。
- **超人飞行**：跳一下就起飞，通过注视飞行，蹲下则降落。使用定位手持控制器，你可以伸出双手到你的两侧滑翔，伸到前方以上升或下降。另外，通过一个小跳跃飞过高层建筑。

- **通过注视钩住**：凝视一个位置，在绳子上弹出一个钩子（或一个蜘蛛网），钩子会固定住而你在空中摆动。
- **第三人称奔跑者**：当你开始快速地移动时，摄像机在你移动时从第一人称视角切换到第三人称视角。当你完成移动时，再切回第一人称视角。
- **飞行员**：你坐在交通工具的驾驶室中，使用各种控制器驾驶。
- **轨道乘行**：在场景中的被动乘行，类似过山车和观光车，没有人控制你去哪里，但是你能够向四周看。
- **注视目标**：场景包括你要凝视的目标，按下按钮，你被传送到场景中的另一个位置。
- **传送门**：通过走入一扇大门或入口实现传送。这是一个常用于在场景和关卡间切换的机制。

你还能想到多少？我们将在后面的章节中实现其中的某几个。

硬件输入设备会很影响我们如何控制我们的第一人称角色。目前，我们受限于现有的硬件。对于本书，我不得不取 Oculus 与 Cardboard 的最低共同标准。所以，我们不能想当然地指定任何输入设备，甚至不能指定简陋的掌上游戏控制器。

然而，在不久的将来，情况将会改变。我们已经有了 **Microsoft Kinect**、**Leap Motion**、**Valve Lighthouse** 和 **Oculus Touch**，以及其他可以实验的设备。随着手持和人体跟踪输入变得更加普遍，开发者将有更多选择在虚拟现实程序中实现移动和传送机制。也许包括以下这些设备：

- **游戏手柄**：现如今，一个传统的游戏控制器、键盘（WASD）或一只鼠标输入设备当然可以用于 VR 程序。问题在于这些设备在你的眼睛被盖住时很难看见该按哪个键。**Xbox One** 的手柄将发布 Oculus Rift 的消费者版本。
- **HMD 触摸板**：三星 Gear VR 在侧面有一个小的触摸板用于用户输入。
- **可定位游戏控制器**：一个包括位置跟踪和动作传感器的手持控制器可以用于掌控你的角色也可以用于手势识别。一些策略性摆放的按钮也可能有效，每只手都有一个。手持控制器的例子是 Valve Lighthouse 和 Oculus Touch。
- **可穿戴运动传感器**：除了你的头部和双手外，如果你在躯干、腿或脚上戴着动作传感器，游戏可以感知所有类型的身体移动。
- **体感摄像头**：我们可以使用红外或其他带有体感识别和跟踪的摄像机技术，比如 Microsoft Kinect。

- **跑步机**：VR 跑步机和相关设备检测你行走的时间和地点。用你的脚和腿四处行走？神奇吧！
- **声音控制**：说出你的想法，取代按钮。
- **手势识别**：使用手势识别，手势输入类型库将会被建立（比如，每个人都知道智能手机上的轻扫和捏合手势。）
- **眼部跟踪**：它检测你的眼球看的方向，并且，也许能识别眼色和其他眼部动作。
- **人脸表情识别，阿尔法脑电波**…… 这个列表还在继续增加。

人们拿基于舌头的输入控制器开玩笑。在 1991 年制作了一个演示文稿嘲笑 *Nose Gesture Interface*（更多信息请参见 http://www.powershow.com/view4/44dcae-ZjBjY/A_Nose_Gesture_Interface_Device_Extending_Virtual_Realities_powerpoint_ppt_presentation）。

当新的输入设备和识别软件浮现于消费者市场时，它们将加速 VR 用户的期待，并且使你的第一人称角色的实现提供最具沉浸感的 VR 体验。

6.6 对付 VR 晕动症

VR 晕动症（motion sickness）或模拟器综合征（simulator sickness），是一种真实的症状并且对于虚拟现实是一个问题。研究人员、心理学家和大量专业技术专家与博士们正在研究这个问题，以便更好地理解底层原因并找到解决方案。

VR 晕动症的一个原因是当你移动你的头部时屏幕更新的卡顿或延迟。你的大脑期待你周围的世界同步的准确变化。至少可以这么说，任何可以感知的延迟都能让你感觉到不适。

延迟可以通过快速渲染每一帧保持建议的帧率而减少。设备厂商比如 Oculus 和其他一些厂商把它当成自己在硬件和设备驱动程序软件中的问题去解决。GPU 和芯片厂商把它作为处理器的性能和吞吐问题。我们将在未来几年中毋庸置疑地见证它们的飞速发展。

同时，作为 VR 开发者，我们需要知晓延迟和 VR 晕动症的其他原因。开发者也需要把它看成我们自己的问题，因为最终它可以归结为性能和人类工程学的问题。对于移动 VR 和桌面 VR 的持续分歧，将一直会存在玩家将要使用到的设备的性能上限。

不仅是技术，我在坐过山车时会感觉到恶心。那么，为什么 VR 不会有类似的效果呢？

对于帮助提高玩家的舒适感和安全感，我们要考虑的事情如下：

- **别移动太快**：当移动或让第一人称角色活动时，别移动太快。高速的第一人称射击游戏在控制台和桌面 PC 游戏上可行但在 VR 中效果不好。
- **向前看**：当在场景中穿过时，如果你看着侧面而不是前方，你更容易感觉到恶心。
- **别太快地转动头部**：不鼓励用户戴着 VR 头盔时快速地转动头部。由于在更短的时间中要在视角中处理更大的变化，HMD 屏幕的更新延迟会加重。
- **提供舒适模式**：当某个场景需要你快速地多次转身时，提供一个棘轮旋转机制，也被称之为舒适模式，让你以更大的增量改变你看的方向。
- **使用一个第三人称摄像机**：如果你有高速动作而你并不需要给用户一种刺激感，使用第三人称视角。
- **着地**：提供视觉提示帮助用户着地，比如横线、视野中的附近的物体和相对固定位置的对象，比如仪表板和人体部位。
- **提供一个重定位视图中心的选项**：尤其是移动 VR 设备，受制于漂移并且偶尔需要被重定位。使用有线的 VR 设备，它帮助你避免在 HMD 线中错乱。作为一个安全性问题，重定位视图中心相关联的真实世界可以帮助你避免撞到物理空间中的家具和墙。
- **别使用过场动画**：在传统游戏（和电影）中，一项可以用于在多个场景中过滤的技术是展示一个 2D 过场动画。如果头部动作捕捉被禁止的话，在 VR 中是行不通的。它会破坏沉浸感并且会引起呕吐。一个替代是简单地淡出到黑色然后打开新的场景。
- **优化渲染性能**：有必要让所有 VR 开发者理解延迟的底层原因，尤其是渲染性能以及你如何优化它，比如降低多边形数量并谨慎选择光照模型。学习使用性能监测工具以便保持帧率在期待和可接受的范围内。第 8 章中将会讨论更多。
- **鼓励用户休息**：另外，你也许可以在游戏中提供一个呕吐袋！

小结

本章中，我们深入讨论了在 Unity 和虚拟现实中拥有一个第一人称角色的意义。我们从取消选择 Unity 的 **Standard Asset** 下的 **Characters** 预制件和它们使用的组件，包括摄像机、角色控制器和 / 或刚体开始。

然后，我们开发了我们自己的第一人称角色，以我们在透视图场景中使用的静态位置的 VR 摄像机装置开始。我们逐渐地增加功能以便顺着你凝视的方向、重力、固体碰撞的方向移动，并使用头部动作开始和停止行走。我们对摄像机高度进行了调整并探索了虚拟现实中第一人称的头部对比身体的关系。最后，我们回顾了一些 VR 开发者应该考虑提供一种舒适的 VR 体验和避免晕动症的因素。

并不是我的意图去说让你应该永远构建你自己的第一人称控制器以代替 Unity 或 VR 集成包中提供的预制件。然而，你当然需要剖析它们中的某一个并自定义它。本章之后，你应该对它可能导致的结果有一个更好的想法。

在第 7 章中，我们将探索刚体的使用、Unity 的物理引擎，以及物理材质，加上一点使用 Blender 进行 3D 建模的内容。

第 7 章 Chapter 7

物理组件和周边环境

跑酷跳跃者：THOR/Parkour Foundations/Flickr, Creative Commons（源自 https://www.flickr.com/photos/geishaboy500/2911897958/in/album-72157607724308547/）

在上一章中，第一人称角色通常限于 *X*-*Z* 平面。本章将更集中在 *y* 轴上，因为我们要探索将物理属性添加到虚拟体验中。你将看到如何把基于物理的属性和材质添加到物体上，以及如何把物理作用力在物体和 C# 脚本之间传递。

在本章中，你将学习以下主题：

- Unity 物理引擎、Unity Rigidbody（刚体）组件和物理材质。
- 用脚本将物理作用力从一个对象传递到另一个上。

- 在第一人称角色上实现速度和重力。
- 与外部互动，包括使用头部射击和跳跃姿势。
- 在 Blender 中构建模型。

注意本章中的项目都是独立的，且不直接与本书中的其他章节关联。如果你决定跳过某些章节或不保存成果物也没有关系。

7.1 Unity 的物理组件

在 Unity 中，一个基于物理的物体的行为是与其网格（形状）、材质（UV 纹理）和渲染属性单独定义的。有关物理的内容项包括以下：

- **刚体（Rigidbody）组件。**
- **碰撞器（Collider）组件。**
- **物理材质（Physic Material）。**
- **项目中的物理管理器（Physics Manager）。**

从根本上来说，物理（本文中的）是由影响物体的位置和旋转的作用力定义的，比如重力、摩擦力、动量，以及与其他物体碰撞产生的力。并不需要完美地模拟真实世界中的物理，因为这是对性能进行的优化以及对关注点的分离，以便实现动画效果。另外，虚拟世界也许就需要它们自己的物理法则，这种法则在上天赐予我们的宇宙中无法找到！

Unity 5 集成了 **NVIDIA PhysX** 引擎，一个实时的物理计算中间件，它为游戏和 3D 应用程序实现了经典牛顿力学。这个多平台的软件已经优化过，在表现效果时可以利用快速硬件处理器。可以通过 Unity 脚本 API 访问。

物理的关键是你添加到物体上的刚体组件。刚体中有一些参数，其中包括重力、质量和阻力。刚体可以自动对重力和与其他物体的碰撞做出反应，而不需要额外的脚本。在游戏中，引擎计算每个刚体的动量，然后更新其位移和旋转。

有关刚体的细节详见 http://docs.unity3d.com/ScriptReference/Rigidbody.html。

Unity 项目中都有一个全局的重力设置，可以通过点击 **Edit | Project Settings | Physics** 出现的用于项目的物理管理器中找到。如你所愿，默认的重力设置值为（0, -9.81, 0）的 **Vector 3**，应用了一个向下的作用力到所有刚体上。重力的单位是 m/s^2。

刚体可以自动对重力和与其他物体的碰撞作出反应，而不需要额外的脚本。

要检测碰撞，发生碰撞的两个物体必须要有碰撞器组件（Collider component）。基本的几何体有内置的碰撞器，比如立体体、球体、圆柱体和胶囊体。网格碰撞器可以假设一个任意的形状。如果可以的话，最好使用一个或多个基本的接近于真实物体的碰撞器形状，而不是使用一个网格碰撞器，以减少在游戏过程中计算真实碰撞的开销。（如果你用了网格碰撞器，网格碰撞器必须被标记为凸面体（convex），且必须小于 255 个三角形。）

当刚体发生碰撞时，在碰撞中与每个物体有关的作用力会被应用到其他物体上。合力的值是基于物体当前的速度和质量计算的。其他因素也会被考虑进来，比如重力和阻力。另外，你可以选择添加约束条件以固定物体在 x，y，z 轴上给的位置值或旋转值。

当物理材质（Physic Material）被指定到物体的碰撞器上时，计算还会被影响更多，会调整摩擦力和碰撞的物体的弹性效果。这些属性只会被应用到拥有物理材质的物体上。（注意，因为某些历史原因，拼写真的是 Physic Material，而不是 Physics Material。）

假如对象 A（球体）碰撞对象 B（砖），如果对象 A 有弹性而对象 B 没有，那么对象 A 在碰撞中将会被施加一个推动力，但是对象 B 没有。然而，你可以选择它们的摩擦力和弹力如何组合，我们后面将会了解。不需要精确地模拟真实世界的物理。它是一个游戏引擎，而不是一个计算机辅助的项目模型。

从脚本的视角来看，当对象碰撞时（OnTriggerEnter）Unity 将触发事件（也叫作消息），对象碰撞时每一帧都触发（OnTriggerStay），当碰撞停止时触发（OnTriggerExit）。

如果上述内容让你听起来有些泄气，继续读下去吧。本章的其余部分会把这些内容拆分成容易理解的小片段。

7.2　弹力球

功能：当一个球体从空中掉落而撞击地面时，它会上下弹跳，不断地被重力减弱。

我们简单地创建一个包含一个地平面和一个球体的新场景。然后，我们添加物理到场景中，一点一点地添加，步骤如下：

1. 通过菜单 **File | New Scene** 创建一个新的场景。

2. 通过菜单 **File | Save Scene As...** 将场景命名为 BallsFromHeaven。

3. 通过菜单 **GameObject | 3D Object | Plane** 创建一个新的平面，通过 **Transform** 组件的齿轮图标 | **Reset** 重置它的位置。

4. 通过菜单 **GameObject | 3D Object | Sphere** 创建一个新的球体，重命名为 Bouncy Ball。

5. 把它的 **Scale** 设置成（0.5, 0.5, 0.5），把 **Position** 设置成（0, 5, 0），让它位于平面的中心之上。

6. 把 **Red** 材质从 Project Assets（第 2 章中创建的）拖到球体上，让它看起来像一个弹力球。

新的 Unity 场景默认带有 **Directional Light** 和 **Main Camera**，暂时使用这个 **Main Camera** 也是可以的。

点击 Play 按钮，什么都没有发生，球体还在半空中并没有移动。

现在，我们给它一个刚体，步骤如下：

1. 选中 BouncyBall，在 **Inspector** 面板中，点击 **Add Component | Physics | Rigidbody**。

2. 点击 Play 按钮，球体会像铅球一样下落。

我们让它变得有弹性，步骤如下；

1. 在 **Project** 面板中，选择 **Assets** 文件夹的根目录，点击 **Create | Folder**，重命名为 Physics。

2. 选中 Physics 文件夹，通过菜单 **Assets | Create | Physic Material** 创建一个材质。

3. 命名为 Bouncy。

4. 把 **Bounciness** 值设置成 1。

5. 在 **Hierarchy** 中选中 BouncyBall 球体，把 Bouncy 资源从 **Project** 面板中拖到球体在 **Inspector** 面板的 **Collider** 材质字段中。

点击 Play 按钮，它会发生弹跳但不会弹到很高。我们给 **Bounciness** 使用的最大值是 1.0。是什么使它变慢的呢？不是 **Friction** 设置，而是设置给 **Average** 的 **Bounce Combine**，它决定了球体（1）与平面（0）的弹力。所以，它的值随着时间迅速变小。我们想让球体仍然有弹力，步骤如下：

1. 把 Bouncy 对象的 **Bounce Combine** 变成 **Maximum**。

2. 点击 Play 按钮。

好多了吧。实际上，好太多了。球体保持弹回到原来的高度，忽略重力了。现在，

把 **Bounciness** 的值变成 0.8，使弹力变小，球体会逐渐停止。

我们在虚拟现实中验证一下，步骤如下：

1. 从 **Hierarchy** 根目录中删除默认的 **Main Camera**。

2. 从 **Project Assets** 中把 MeMyselfEye 预制件拖进场景，把 **Position** 值设置成 (0, 1, -4)。

在虚拟现实中运行，相当不错！甚至最简单的东西在虚拟现实中也看起来很棒。

好了，我们来玩点有趣的，让球体像雨一样落下！要实现这样的效果，我们要把球体制作成预制件，再写一个脚本实例化新的球体，让它们在随机的位置下落，步骤如下：

1. 把 BouncyBall 从 **Hierarchy** 中拖进 **Project** 的 Assets/Prefabs 文件夹，让它成为预制件。

2. 在 **Hierarchy** 中删除 BouncyBall，因为我们要用代码实例化它。

3. 通过菜单 **GameObject | Create Empty** 创建一个空的游戏控制器对象以附加脚本，重命名为 GameController。

4. 在 **Inspector** 中，点击 **Add Component | New Script**，命名为 BallsFromHeaven，在 MonoDevelop 打开脚本。

像下面这样编辑脚本：

```
using UnityEngine;
using System.Collections;

public class BallsFromHeaven : MonoBehaviour {
  public GameObject ball;
  public float startHeight = 10f;
  public float fireInterval = 0.5f;

  private float nextBallTime = 0.0f;

  void Update () {
    if (Time.time > nextBallTime) {
      nextBallTime = Time.time + fireInterval;
      Vector3 position = new Vector3( Random.Range (-4.0f, 4.0f),
        startHeight, Random.Range (-4.0f, 4.0f) );
      Instantiate( ball, position, Quaternion.identity );
    }
  }
}
```

这段脚本以 `fireInterval` 的速度（0.5 的间隔意味着每半秒就有一个新的球体掉

落）从 startHeight 掉落一个新的球体。新的球体掉落的位置是 ***X-Z*** 坐标中 -4 到 4 的值。Instantiate( ) 函数添加一个新的球体到场景的 **Hierarchy** 中。

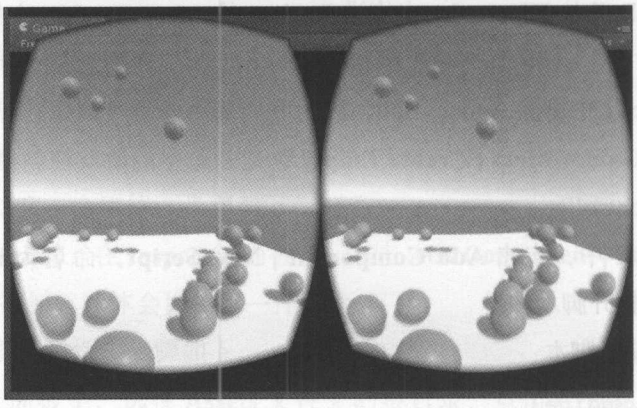
保存脚本。我们需要把 **Ball** 字段放进 BouncyBall 预制件中，步骤如下：

1. 在 **Hierarchy** 中选中 GameController，把 BouncyBall 预制件从 **Project** 的 Assets/Prefabs 文件夹拖到 **Inspector** 中的 **Balls From Heaven（Script）** 面板的 **Ball** 槽中。

2. 确认使用的是 **Project Assets** 中的 BouncyBall 预制件，让它可以被实例化。

3. 保存场景。在虚拟现实中运行。有意思吧！

这是我们看到的：



当球体被实例化时观察 **Hierarchy** 面板，注意有些球体在平面上停止弹跳了但仍然还在 **Hierarchy** 面板中，我们需要添加一个脚本来清理这些球体，这个脚本会销毁那些完成任务的球体。步骤如下：

1. 在 **Project** 面板的 Assets/Prefabs 中选择 BouncyBall 预制件。

2. 点击 **Add Component | New Script**，命名为 DestroySelf，在 MonoDevelop 中打开。

这是 DestroySelf.cs 脚本，当球体的 Position 的 *Y* 值在地下面（***Y*=0**）以下时脚本会销毁它。

```
using UnityEngine;
using System.Collections;

public class DestroySelf : MonoBehaviour {
  void Update () {
    if (transform.position.y < -5f) {
      Destroy (gameObject);
```

```
        }
    }
}
```

无论什么时候你的脚本实例化对象时，你都必须清楚这个对象的生命周期，并尽可能地在不需要它时把它销毁。

总结一下，我们创建一个带有 Rigidbody 的球体，并添加一个带有值为 0.8 的 **Bounciness** 属性的 Physic Material，**Bounce Combine** 为 **Maximum**。然后，我们把 BouncyBall 变成了一个预制件，并编写了一个实例化从上面掉落的新球体的脚本。

7.3　用头部射击

真正来玩这些弹力球不是很有意思吗？让我们制作一个游戏——试着用头部动作把球体当作目标瞄准。对于这个游戏，球体每次从上面下落并弹到你的前额（脸部）高度，把它作为目标瞄准。

功能：当球体从你的头部上面下落时，你把它弹回到脸部并把它作为目标瞄准。

要实现这个游戏，把一个立方体碰撞器放在 MeMyselfEye 头上，VR 摄像机作为其父结点，让我们的头部动作能够移动立方体的表面。我认为对于这个游戏来说，立方体形状的碰撞器会比球体和胶囊体要好，因为它提供了一个平的表面（像一只划桨），让弹力的方向更准确。球体将从天空中消失。对于一个目标，我们将使用一个平整的圆柱体。我们还将添加声音提示新的球体已经释放，以及提示球体击中了目标。

创建一个新的场景，然后按下面的步骤实现这个头部：

1. 点击菜单 **File | Save Scene as**，命名为 BallGame。

2. 使用齿轮图标 **Remove Component** 删除附加到 GameController 的 **BallsFromHeaven** 脚本组件，我们不再需要它了。

3. 在 **Hierarchy** 面板中，展开 MeMyselfEye 钻取到 **Main Camera** 对象，并选中它。

4. 在 **Inspector** 面板中，点击菜单 **Add Component | Physics | Box collider**。

5. 选中 GameController，点击菜单 **Add Component | Audio | Audio Source**。

6. 点击 **Audio Source** 的 **AudioClip** 字段最右边的小圆形图标以打开 **Select AudioClip** 对话框，然后选择名为 Jump 的片段。

7. 选中 GameController，点击菜单 **Add Component | New Script**，命名为 BallGame，在 MonoDevelop 中打开。

注意 **Scene** 视图中的盒子碰撞器，它包围了你的第一人称角色的 head/camera。我们将播放这个 Jump 声音片段（Unity 的 **Standard Assets** 的 **Characters** 包已经提供）以提示新的球体下落了。

下面是 BallGame.cs 脚本：

```
using UnityEngine;
using System.Collections;

public class BallGame : MonoBehaviour {
  public GameObject ball;
  public float startHeight = 10f;
  public float fireInterval = 5f;

  private float nextBallTime = 0.0f;
  private GameObject activeBall;
  private Transform head;
  private AudioSource audio;

  void Start () {
    head = Camera.main.transform;
    audio = GetComponent<AudioSource> ();
  }
  void Update () {
    if (Time.time > nextBallTime) {
      nextBallTime = Time.time + fireInterval;
      audio.Play ();
      Vector3 position = new Vector3( head.position.x,
        startHeight, head.position.z + 0.2f );
      activeBall = Instantiate( ball, position,
        Quaternion.identity ) as GameObject;
    }
  }
}
```

我们每 `fireInterval` 秒实例化一个新的球体，位置值是 `strtHeight`，在当前的 head 位置之上，且稍微靠前 `(0.2f)` 一点，让它不会直接掉落在我们的头顶上。没有旋转值应用给球体 `(Quaternion.identity)`。

在 Unity 编辑器中放置一个公有变量，步骤如下：

1. 选中 GameController，把 BouncyBall 从 **Project Assets** 拖到 **Inspector** 中的 **Ball Game** 面板的 **Ball** 字段上。

2. 在虚拟现实中试着运行。

当你听见球体时，向上看并以某个角度瞄准你面前弹起的球体。棒！

现在，我们需要目标，执行下面的步骤：

1. 为目标创建一个平的圆柱体，使用菜单 **GameObject | 3D Object | Cylinder**，命名为 Target。

2. 把它的 **Scale** 设置为（3, 0.1, 3），把 **Position** 设置为（1, 0.2, 2.5），让它位于你面前的地面上。

3. 把 Blue 材质（第 2 章中创建的）从 Project Assets/Materials 文件夹拖到它上面。

4. 注意，它默认的胶囊碰撞器是半球形的，这样恐怕不行。在 Capsule Collider 上，选择它的齿轮图标 | **Remove Component**。

5. 点击菜单 **Add Component | Physics | Mesh Collider**。

6. 在新的 **Mesh Collider** 中，勾选 **Convex** 复选框和 **Is Trigger** 复选框。

7. 使用菜单 **Add Component | Audio | Audio Source** 添加一个声音源。

8. 选中 Target，点击 **AudioClip** 字段最右边的小圆形图标，以打开 **Select AudioClip** 对话框，然后选择名为 Land 的片断。

9. 添加一个新的脚本，点击菜单 **Add Component | New Script**，命名为 TriggerSound，在 MovoDevelop 中打开。

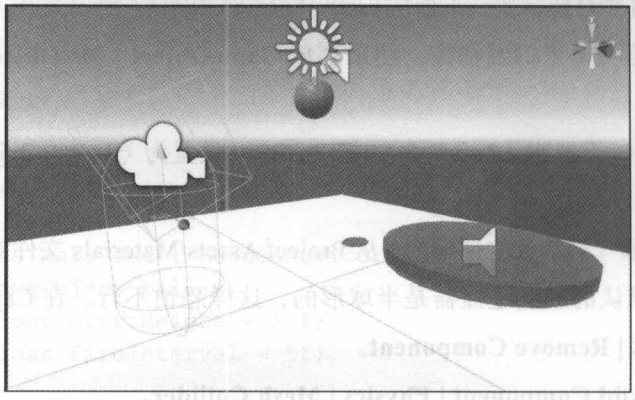
当你击中目标时，下面的 TriggerSound.cs 脚本将播放一段声音片断：

```
using UnityEngine;
using System.Collections;

public class TriggerSound : MonoBehaviour {
  public AudioClip hitSound;

  void Start() {
    audio = GetComponent<AudioSource> ();
  }

  void OnTriggerEnter(Collider other) {
    audio.Play ();
  }
}
```

这个脚本使用 `OntriggerEnter()` 消息处理器以得知何时播放声音片断。我们再次利用了 Unity 的 **Standard Assets** 中的 **Characters** 包中的 Land 片断。

在 VR 中试运行，它是一个 VR 游戏！下图展示了这个带有第一人称碰撞器和一个

球体从立方体碰撞体朝着目标弹出的场景：



额外挑战：保存得分。提供一个瞄准十字星，添加一个背景板以及一些其他功能让游戏更有挑战性。比如，你可以改变开火间隔或者增加球体的速度。

到现在，我们通过一个 Physic Material 把 **Bounciness** 附加到一个球体对象上。当球体碰撞另一个对象时，Unity 的物理引擎将计算弹力以确定球体的新的速度和方向。在下一节中，我们将看一下如何把弹力从一个对象传递给另一个对象。

7.4 蹦床与弹力球

蹦床与弹力球的不同之处在于前者与物体碰撞之后物体被弹起而不是让自己弹起。Unity 并没有帮我们实现它，所以我们需要使用脚本。

功能：当一块砖从半空中掉落到一个蹦床上时，它会弹起并因重力而减弱。

构建一个场景并把目标变成一个蹦床，步骤如下：

1. 点击菜单 **File | Save Scene As** 并命名为 BrickTrampoline。

2. 从 GameController 中使用齿轮图标 | **Remove Component** 删除 BallGame 脚本组件，我们不需要它了。

3. 将 Target 对象重命名为 Trampoline。

4. 将其 **Position** 设置为（0, 0.2 ,0）。

5. 创建砖块，点击菜单 **GameObject | 3D Object | Cube**，并命名为 Brick。

6. 设置其 **Scale** 为（0.25, 0.5, 1），**Position** 为（0, 5, 0）。

7. 把 Red 材质拖到它上面。

8. 通过菜单 **Add Component | Physics | Rigidbody** 添加一个刚体。

当你运行时，砖块掉落后会完全停止。在蹦床上创建一个新的脚本，步骤如下：

1. 在 **Hierarchy** 中选中 Trampoline，通过菜单 **Add Component | New Script** 创建脚本。

2. 将脚本命名为 Trampoline，打开编辑。

Trampoline.cs 脚本如下：

```
Using UnityEngine;
using System.Collections;

public class Trampoline : MonoBehaviour {
  public float bounceForce = 1000.0f;

  void OnTriggerEnter( Collider other ) {
    Rigidbody rb = other.GetComponent<Rigidbody> ();
    if (rb != null) {
      rb.AddForce (Vector3.up * bounceForce);
    }
  }
}
```

当刚体对象碰撞蹦床时，`OnTriggerEnter()` 函数添加一个 `bounceForce` 到它的刚体上。

保存场景，运行 VR，砖块现在会在蹦床上弹起来。你可能需要调整 **Bounce Force** 值，增加或减少。

总结一下，我们创建一个带有刚体的砖块、一个不带刚体的蹦床以及一个向上的碰撞器，蹦床给砖块添加一个向上的力。

7.5　人类的蹦床

现在，你可以自己在蹦床上蹦跳。

功能： 当第一人称角色撞击蹦床时，它会弹起来并因重力而减弱。

7.5.1　像一块砖

实现这个功能的一种方式是把 MeMyselfEye 第一人称角色当成一块砖，并给它一个刚体（Rigidbody）和一个胶囊碰撞器（Capsule Collider），让它能够使用物理进行响

应。我们会先试一下这种方式看看它是否可行。若想让这种方式运行，我们需要禁用它的角色控制器组件，并让它出现在蹦床之上的砖块的位置，以便我们能够下落，步骤如下：

1. 点击菜单 **File | Save Scene As**，命名为 HumanTrampoline。

2. 在 **Hierarchy** 中删除 Brick，我们不再需要它了。

3. 在 **Hierarchy** 面板中，选中 MeMyselfEye，将其 **Position** 设置为（0, 0.5, 0）。

4. 点击菜单 **Add Component | Physics | Rigidbody**。

5. 点击菜单 **Add Component | Physics | Capsule Collider**，将其 **Height** 设置为 2。

6. 在 **Rigidbody** 面板中，**Constraints** 的下方，反选 **Freeze Rotation X, Y, Z** 复选框，这样我们就不会头晕了。

运行场景，哇哦！我们正在跳起和下落。你可能需要调整 **Bounce Force** 值。

Trampoline 脚本调用刚体的带有一个 bounceForce 参数的 AddForce( ) 函数。然而，角色没有移动其自身。我们可以继续走这条路，但是我们不这么做。

7.5.2 像一个人物角色

在上一章中，我们给了第一人称 MeMyselfEye 一个角色控制器组件，因为它给我们带来很多很棒的功能，这些功能是第一人称角色（包括一个碰撞器和玩家友好的物理属性）所需要的。我们想再次使用它，从创建一个新的 MeMyselfEye 的复本开始，步骤如下：

1. 在 **Hierarchy** 面板中，删除现有的 MeMyselfEye 对象。

2. 从 **Project** 面板中的 Assets/Prefabs 文件夹里，把 MeMyselfEye 预制件拖进场景中。

3. 将其 **Position** 设置成（0, 1, –4）。

4. 点击菜单 **Add Component | Physics | Character Controller**。

5. 通过菜单 **Add Component | New Script** 添加 HeadLookWalkBounce 脚本。

这个脚本与第 6 章中编写的 HeadLookWalk 脚本类似，但是略微有些不同。这一次，我们需要自己实现大多数的物理特性。这意味着不再使用 CharacterController.SimpleMove( )，我们将使用更灵活的 CharacterController.Move( )。SimpleMove 忽略移动方向中的 y 轴，但是我们需要对弹力应用它。

打开 HeadLookWalkBounce.cs 脚本并像下面这样编辑：

```
using UnityEngine;
using System.Collections;

public class HeadLookWalkBounce : MonoBehaviour {
  public float velocity = 0.7f;
  public bool walking = false;

  public float gravity = 9.8f;
  public float bounceForce = 0.0f;

  private CharacterController controller;
  private Clicker clicker = new Clicker();
  private float verticalVelocity = 0.0f;
  private Vector3 moveDirection = Vector3.zero;

  void Start() {
    controller = GetComponent<CharacterController> ();
  }

  void Update () {
    if (clicker.clicked()) {
      walking = !walking;
    }
    if (walking) {
      moveDirection = Camera.main.transform.forward * velocity;
    } else {
      moveDirection = Vector3.zero;
    }
    if (controller.isGrounded) {
      verticalVelocity = 0.0f;
    }
    if (bounceForce != 0.0f) {
      verticalVelocity = bounceForce * 0.02f;
      bounceForce = 0.0f;
    }
    moveDirection.y = verticalVelocity;
    verticalVelocity -= gravity * Time.deltaTime;
    controller.Move (moveDirection * Time.deltaTime);
  }
}
```

这段脚本不仅管理横向速度，还管理从 bounceForce 和重力计算出的 verticalVelocity。如果你正站在任何固体之上（isGrounded），verticalVelocity 的值会被设置成 0。如果你在空中，你将不再着地，并且重力将会被施加。

修改 Trampoline.cs 脚本，发送 bounceForce 给角色的 HeadLookWalk 脚本组件，如下：

```
using UnityEngine;
using System.Collections;

public class Trampoline : MonoBehaviour {
  public float bounceForce = 300f;

  void OnTriggerEnter( Collider other ) {
    Rigidbody rb = other.GetComponent<Rigidbody> ();
    if (rb != null) {
      rb.AddForce (Vector3.up * bounceForce);
    } else {
      HeadLookWalkBounce locomotor =
        other.GetComponent<HeadLookWalkBounce> ();
      if (locomotor != null) {
        locomotor.bounceForce = bounceForce;
      }
    }
  }
}
```

蹦床现在可以处理来自角色或非角色对象的碰撞。砖块和角色对 bounceForce 的反应不同。所以，神奇的作用力因子（forceFactor）使它们相等（试着调整蹦床中的这个值和 / 或 bounceForce）。

在 VR 中运行，当检测到碰撞时向前走到蹦床之上。你将会垂直地飞起来然后又落下去。

这并不是蹦床的机制，因为你只是挨着而并没有在上面跳动就被推到空中。但是，它适用于我们的目的。

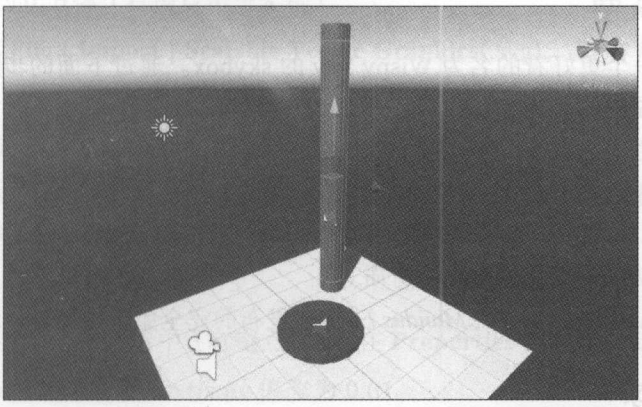
为了有趣，创建一个墩子靠近蹦床，并试着用下面的步骤降落在墩子上。

1. 点击菜单 **GameObject | 3D Object | Cylinder**，并重命名为 Pillar。

2. 设置其 **Position** 为（–2, 5, 2.2），**Scale** 为（1, –5, 1）。

3. 把名为 Red 的材质拖到它上面。

4. 保存场景并在 VR 中运行。哇哦！注意我们跳到多高。当在空中时，朝墩子看并降落在它上面。如果你继续从上面走下去会掉落在地面上。这就是它看起来的样子：



总结一下我们所做的：

- 首先，我们创建了一个带有物理材质的 BouncyBall，当碰撞时 Unity 自动把这个物理材质应用到球体上，而不需要写脚本。
- 然后，我们创建了一块砖，它受到一个向上的作用力，来自于 Trampoline 脚本中使用 Rigidbody.AddForce( )。
- 最后，在前一节中，第一人称角色的脚本通过蹦床设置它的公有变量 bounceFoce，手动地使用 CharacterController.Move( ) 把它作为垂直方向的速度与重力一起施加。

这个带有物理材质的 BouncyBall 对象会通过 Unity 的物理引擎自动移动而不用写脚本。这块砖会通过蹦床移动，直接添加一个作用力到砖块的刚体上。这个第一人称角色可以通过其基于重力和蹦床设置的 bounceForce 变量计算得出的自己移动的方向而移动。

7.6　插曲——环境和万物

让我们用更有意思的周边环境和几何对象给这个场景增添一些乐趣。本节为可选章节。你可以按照下面的步骤或者就按照自己的想法去做。

我已经在下载文件中提供了与本书相关的资源。在后面的章节中会提及，这些资源中的一些可能会作为免费包的一部分，可以在 Unity 的 **Asset Store** 中找到，并且已得到创建者的许可：

1. 创建场景的一个新的版本，通过菜单 **File | Save Scene As...** 保存并命名为 PhysicsWorld。
2. 选择地平面并把它的 **Scale** 改成（5, 1, 5）。

现在，我们来添加天空和地球。

7.6.1 缥缈的天空

我们将添加一个更好看的名为 Wispy Sky 的 skybox。通过下面的步骤添加 skybox：

1. 导入本书提供的名为 WispySky.package 资源包。
2. 在主菜单栏中，点击 **Window | Lighting**。
3. 在 **Lighting** 面板中，选择 **Scene** 选项卡。
4. 在 **Skybox** 字段中，点击最右边的圆形图标打开 **Select Material** 对话框。
5. 选择名为 WispySkyboxMat 的材质。

你还可以把 **Asset Store** 中由 *Mundus Limited* 发布的完全免费的 Wispy Skybox 包拿来用。

7.6.2 行星地球

添加一个地球仪。我们可以用下面的步骤来完成：

1. 导入本书提供的名为 PlanetEarth.package 的资源包。
2. 在 **Project** 面板中，钻取到下面找到 Earth3968Tris 预制件，把它拖进 **Scene** 视图中。
3. 把它的 **Position** 设置成（100, 0, 300）。
4. 把它的 **Scale** 设置成（10, 10, 10）。
5. 把 EarthOrbit 动画片断拖到它之上。
6. 在 **Hierarchy** 中展开 Earth3968Tris 并选择其同名子对象，把这个对象的 **Rotation** 设置成（0, 90, 340）。

你还可以把 **Asset Store** 中由 Close Quarter Games 发布的完全免费的 Planet Earth Free 包拿来用。纹理出处在 http://www.shadedrelief.com/natural3/pages/textures.html。

7.6.3 企业标识

接下来，我们为什么不用一些对你来说有意义的东西来个性化这个场景呢？我们要让它很大，然后跳到上面。它可能是一把吉他、一个“小马宝莉”玩具，或者只是一堆 Unity 中基础的 3D 对象。我要使用 *Packt* 的商标，因为他们会出版这本书，我们将使用 Blender 来完成。

7.6.3.1 Blender

因为我们需要一个矢量图形版本的商标，我用了一个 PNG 图片文件，已经在 Gimp 中剪裁过，上传到 Vector Magic（http://vectormagic.com/）并得到一个 SVG 文件。源文件都已经包含在本书中了。然后，我在 Blender 中把 2D 图片转换成 3D 模型，再简单地挤压它一次，可通过下面的步骤实现：

1. 打开 Blender 应用程序，选择 **New** 文件。

2. 删除默认的立方体（右键点击＋X 键）。

3. 点击菜单 **File | Import | Scalable Vector Graphics**(.svg)，然后加载 Packt_Logo1.svg。

4. 把视图变成 **Top Ortho**（小键盘 1＋5）。

5. VectorMagic 包含一个用于背景的对象，选择这个对象然后将其删除（右键点击＋X 键）。

6. 全部选择（A 键）。

7. 对于每个字母，将其选中（右键点击）。在 **Properties** 面板中，选择数据图标选项卡。在 **Geometry** 面板下，把 **Extrude** 值改为 0.01。

8. 保存文件为 logo.blend。

7.6.3.2　Unity

在 Unity 中，我们执行下面的步骤：

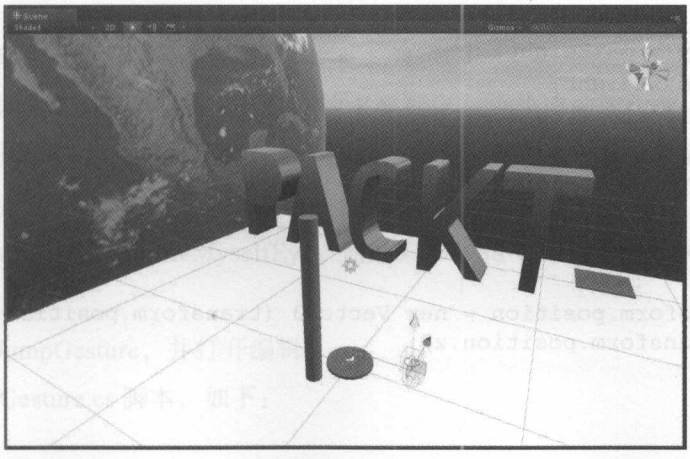
1. 把 logo.blend 文件拖进 **Project** 面板的 Assets/Models 文件夹。

2. 配置 **Import Settings** 的值，**Scale Factor** 值为 10，**Mesh Compression** 的值为 **High**，选中 **Generate Colliders**，反选 **Import Materials**。

3. 缩放与位置按你的喜好来定，我把 **Position** 设置成（18, 11, 24），**Rotation** 设置成（90, 270, 0），**Scale** 设置成（20, 20, 20）。

4. 我用了一个金属材质将其润色。创建一个新的材质，命名为 ShinyMetalic，把它的 **Metalic** 值设置成 1，把 **Smoothness** 设置成 0.8。把它拖到商标的每个字母上。

下图展示了我的场景：



7.7 升降机

想从商标的顶部看这个视图吗？让我们制作一个马里奥（Mario）风格的升降机到达顶部。

功能：提供一个升降机平台，它可以上下移动，让我能够在上面行走并乘坐它到商标的顶部。

通过下面的步骤把它构建进场景中：

1. 通过菜单 **GameObject | 3D Object | Cube** 创建一个升降机平台，命名为 Elevator。

2. 设置其属性让它与商标对齐。通过设置属性让你可以乘坐在平台上，然后走出平台到达商标顶部。我把 **Position** 设置成（17, 1.4, –8.8），把 **Scale** 设置成（4, 0.1, 4）。

3. 把 Blue 材质拖到它之上。

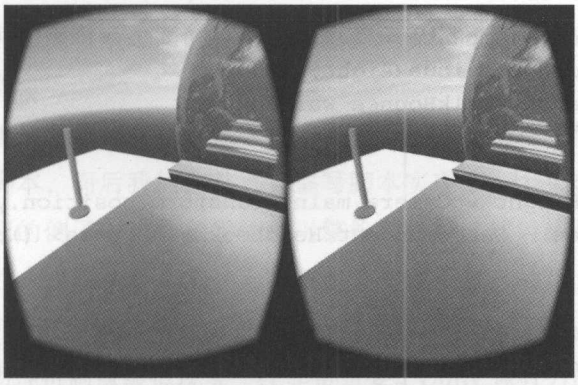
4. 通过菜单 **Add Component | New Script** 创建脚本，命名为 Elevate，并打开进行编辑。

下面是 Elevate.cs 的代码：

```
using UnityEngine;
using System.Collections;

public class Elevator : MonoBehaviour {
  public float minHeight = 1.2f;
  public float maxHeight = 8.0f;
  public float velocity = 1;

  void Update () {
    float y = transform.position.y;
    y += velocity * Time.deltaTime;
    if (y > maxHeight) {
    y = maxHeight;
      velocity = -velocity;
    }
    if (y < minHeight) {
      y = minHeight;
      velocity = -velocity;
    }
    transform.position = new Vector3 (transform.position.x, y,
      transform.position.z);
  }
}
```

这段脚本简单地在每一帧中上下移动这个平台。没有物理组件，只有简单地动画。但是，它是一个刚体，而且我们可以站在它之上。

在 VR 中运行，试着走上升降机乘坐到顶部，然后移动到这个对象的结构的顶部。下图展示的是从顶部看我的视图：



7.8　跳起来

当我们创建蹦床时，我提到我们的实现不需要你跳上去才开始弹。然而，跳起来是一个有意思的想法，不管在不在蹦床上。有些游戏使用键盘的空格键或一个控制器的按键让虚拟角色跳起来。我们现在要在 VR 头盔中实现一个简单的跳跃姿势，这将会有一个垂直的速度应用到我们的位移中。

注意，如果头盔中缺少位置跟踪器将实现不了这个想法，比如 Google Cardboard 和 GearVR。

功能：当我跳跃时，我的角色在 VR 中也跳跃。

要实现跳跃的姿势，我们要找到头盔的 ***Y*** 坐标上一个快速的变化值，非常像我们在第 5 章中（检测 *x* 轴角度上的快速变化值）实现的点头动作。当检测到跳跃动作时，我们将应用一个垂直方向的作用力到第一人称角色上，步骤如下：

1. 在 **Hierarchy** 中选择 MeMyselfEye，通过菜单 **Add Component | New Script** 创建脚本。

2. 命名为 JumpGesture，并打开编辑。

编辑 JumpGesture.cs 脚本，如下：

```
using UnityEngine;
using System.Collections;

public class JumpGesture : MonoBehaviour {
  public bool isJump = false;
  public float jumpForce = 1000.0f;

  private float jumpRate = 1.0f;
  private float previousHeight;
  private HeadLookWalkBounce walkBounce;

  void Start () {
    previousHeight = Camera.main.transform.position.y;
    walkBounce = GetComponent<HeadLookWalkBounce> ();
  }

  void Update () {
    if (DetectJump ()) {
      walkBounce.bounceForce = jumpForce;
    }
  }

  private bool DetectJump() {
    float height = Camera.main.transform.localPosition.y;
    float deltaHeight = height - previousHeight;
    float rate = deltaHeight / Time.deltaTime;
    previousHeight = height;
    return (rate >= jumpRate);
  }
}
```

Update() 函数调用 DetectJump，通过检测摄像机 *Y* 坐标上的快速变化值以决定玩家是否在真实世界中进行跳跃。如果玩家确实跳跃了，那么在 HeadLoookWalk-Bounce 脚本中设置 bounceForce，就像是蹦床所做的。如果你愿意，你还可以修改 jumpForce，让它与蹦床使用的值不同。

在 VR 中运行一下。哈！看，你不需要烦人的游戏控制器就可以跳起来了！使用你的股四头肌。

这是一个简单的用于阐明目的的近似法，只看前一帧的运动变化。我鼓励你探索新的更好的方法来使用头部和身体作为 VR 的输入。

额外挑战：对于不带位置跟踪的移动 VR 设备，尝试其他的方式引起跳跃。

小结

本章中，我们体验了一次 Unity 物理引擎的豪华旅行。首先，我阐述了一些外行术语：刚体、碰撞器和物理材质之间的关系，以及物理引擎如何使用这些东西确定速度和场景中物体间的碰撞。

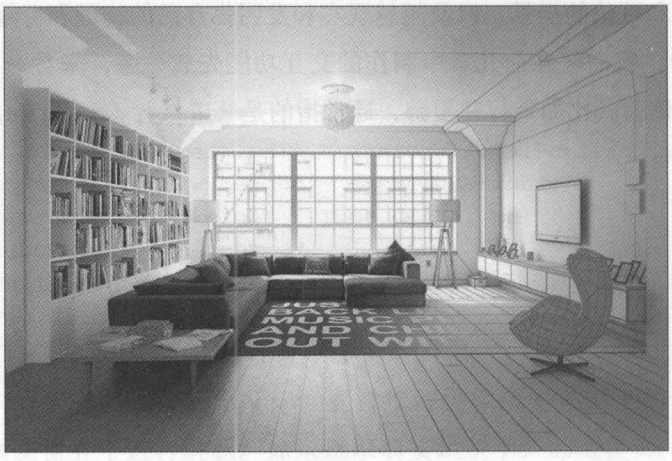
然后，我们详细地学习了各种直接、间接和完全不使用物理引擎的例子。弹力球使用了引擎而不用写脚本，而后我们在其之上编写脚本实现了一个头盔游戏和一个淋浴般的球体下落流。蹦床的例子中我们使用物理引擎检测碰撞，然后编写脚本将作用力传递给另一个对象。最后，我们在一个第一人称角色上实现了自己的重力和弹力，其中包括一个跳跃动作。我们在环绕地球飞行时完成了所有这些！奇迹永无止境！

在游戏设计中物理机制像虚拟现实一样非常重要。Unity 强有力的物理 API 给开发者构建了相当准确和可信的场景，以及超越现实和发明你自己的物理和奇趣所需要的工具。

在下一章中，我们将在前几章中制作的交互功能先放一旁，来看一些较为封闭或被动、动态的 VR 体验，比如乘行并漫游，通常指的是轨道乘行（riding on rails）。

Chapter 8

第 8 章
漫游和渲染



客厅模拟场景，Krystian Babilinsky，已授权

本章中，我们将稍微深入场景设计、建模、渲染并实现一个可以在 VR 中体验的漫游动画。场景是一个画廊，在场景中你设计一个简单的地板平面图并使用 Blender 纵向延伸到墙上。使用你自己的照片，然后简单地浏览一下。最后，我们会有一个关于优化、性能和舒适度的技术交流。

本章中，我们将讨论以下话题：

1. 使用 Blender 和 Unity 构建一个简单化的画廊。

2. 创建一个画廊场景的漫游动画。

3. 复杂场景中可以用于获取性能的技术。

注意，本章中的项目都是独立的并不依赖于本书中其他章节中的项目。如果你决定跳过其中的一些或者不保存成果，也无所谓。

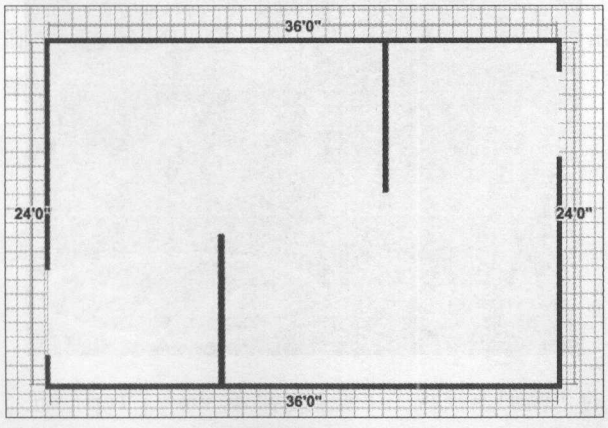
8.1　用 Blender 构建

使用令人信服的材质和光照创建逼真的模型是一门艺术，也是一门超出本书范围科学。很多人直接在网上寻找专业人员制作的模型，包括 Unity Asset Store。当然了，这里有各种各样的设计类程序，从最高级的，比如 3D Studio Max（http://www.autodesk.com/products/3ds-max/）和 Blender，到最容易使用的，比如 Home Styler（http://www.homestyler.com/）和 SketchUp（http://www.sketchup.com/）。

对于本项目，我们只需要一些简单的东西。我们需要一个小的艺术相册展览室，大约为 7.3m×11m。保持其简单且具有启发性，我想向你们展示如何在 Blender 中开始建模。

8.1.1　墙体

开始之前，在一张纸上或通过一个绘图程序画一幅简单的建筑平面图。我的平面图呈现的是一个有两个入口和一些用于显示艺术品（Gallery-floorplan.jpg）的室内墙体的开放空间，看起来就像下面的图片：



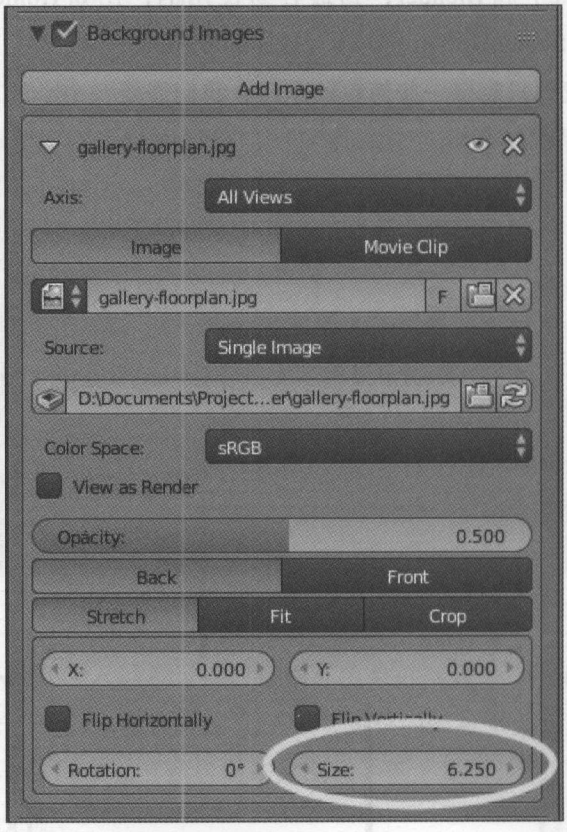
现在，打开 Blender。我们将用一个常用的技巧创建一个简单的对象（一个平面），再

拉伸制成墙体的每一面。要实现这些，执行下面的步骤：

1. 建立一个空白场景，按 A 键全选，再按 X 键删除。

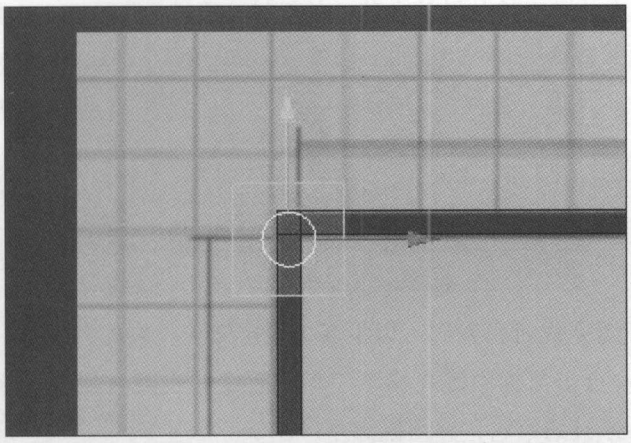
2. 添加用于参考和打印的建筑图，按 N 键打开属性面板。在 **Background Images** 面板中，选择 **Add Image**，点击 **Open**，再选择你的图片（Gallery-floorplan.jpg）。根据建筑平面参考图的尺寸和比例尺，你需要选择一个比例因子让它正确地出现在 Blender 世界坐标空间中。6.25 的缩放比例对我来说刚好。实际上，最重要的事情是图表中的功能的相对比例，因为我们可以一直在 Unity 的 **Import** 设置中调整缩放比例，或者在 **Scene** 视图中调整。

3. 在 **Background Images** 面板中，将 **Size** 设置成 6.25。这个面板，**Size** 字段已经高亮，如下面的截图所示：



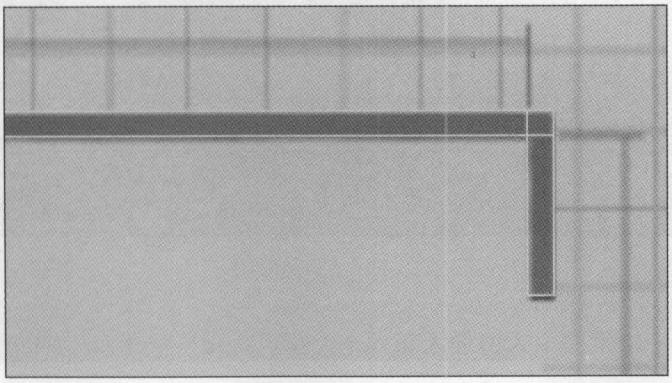
4. 按数字盘上的 7 键到顶级视图（或点击菜单 **View | Top**），按 5 键到正交视图（或点击菜单 **View | Persp/Ortho**）。注意背景图片只会在 **top-ortho** 视图下被绘制。

5. 按 Shift＋A 键添加一个面板。选择这些键，按 Tab 键进入 **Edit** 模式。按 Z 键从实体图切换到 **wireframe**（线框图）。按 G 键把它拖到一个角上，再按回车键确认。按 S 键缩放它到适合墙的宽度，如下面的截图所示（你可能会想起可以使用鼠标滚轮来进行缩放，以及用 Shift 键和点击鼠标中键移动）：

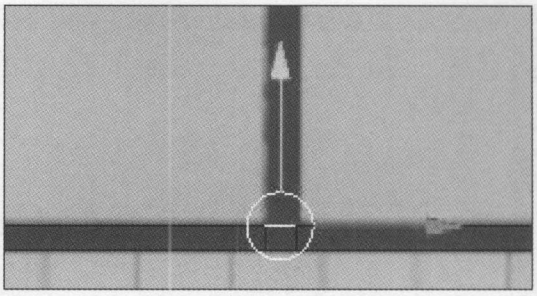


6. 现在，我们要拉伸它来制作外墙。进入 **Edge Select** 模式（通过下面截图中的图标），按 A 键反选所有，再在你要拉伸的边上点击右键。按 E 键开始拉伸，按 X 或 Y 键把它约束在轴上，再在需要的时候按回车键完成拉伸：

7. 对每面外墙重复前面的步骤。在各个角创建一个方块，这样你就可以在垂直方向上进行拉伸。给门廊留些空隙。你可能会想要修改现有的边，请点击右键选择需要改的边，Shift＋右键选择多个，用 G 键移动。你可以复制选中的项：

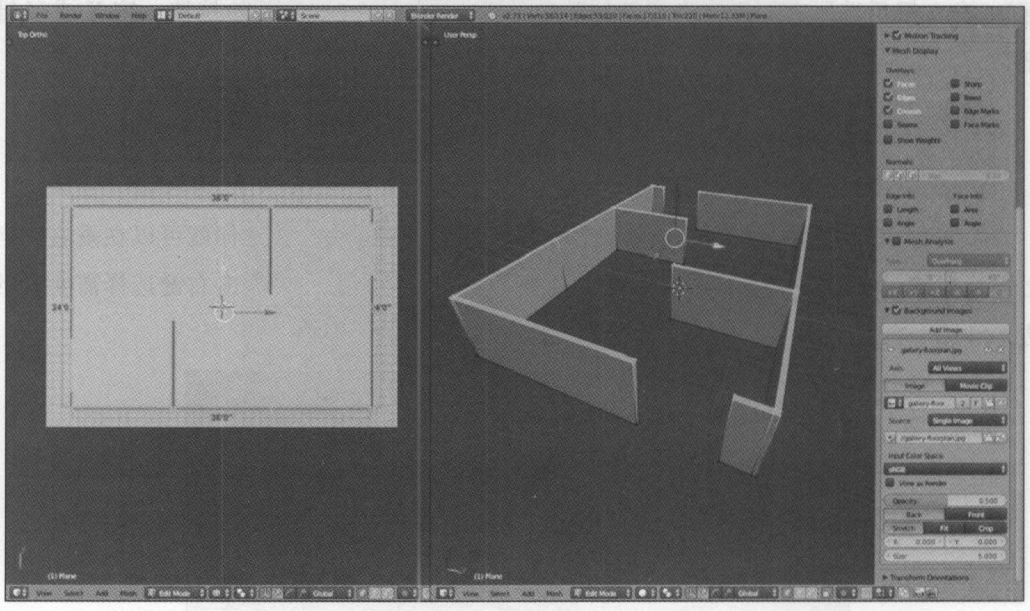


8. 要想从中间拉伸一个面，我们需要添加一个边缘的循环。鼠标移至这个面上，按 Ctrl+R 键再左键创建一个切口。滑动鼠标以定位，再点击鼠标左键确认。对这些墙的宽度重复这些步骤（在外墙制作一个方块切口）。选择这条边再按 E 键把它拉伸进房间：



9. 当建筑平面图完成时，我们可以沿着 *z* 轴拉伸它以创建各面墙。按 5 键把视图从 **Ortho** 变成 **Persp**。点击鼠标中键并移动使其向后倾斜。按 A 键全选，用 E 键拉伸。用鼠标开始拉伸，按 Z 键约束它，并用鼠标左键确认。

10. 把模型保存到文件并命名为 gallery.blend：



8.1.2 天花板

现在，添加一块带两个天窗的天花板。天花板将是一块来自一个立方体的平板。让

我们看看添加天花板的步骤：

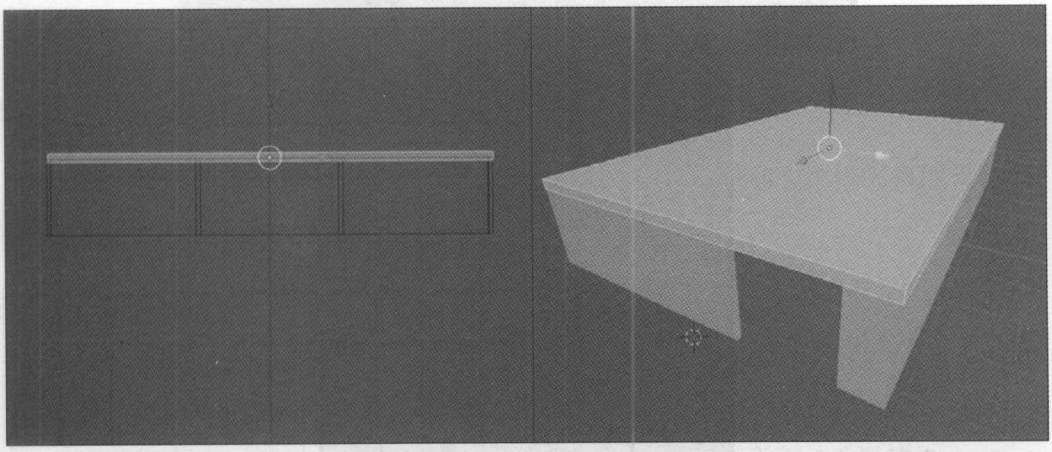
1. 用 Tab 键返回 **Object** 模式。

2. 用 Shift＋A 键创建一个立方体。

3. 用 G 键把它置于中央。

4. 用 S＋X 和 S＋Y 沿着 x 轴和 y 轴缩放，使它的尺寸与房间的大小相同。

5. 用 1 键切换到 **Front** 视图，用 S＋Z 把它缩放成平的，用 G＋Z 把它移动到墙的顶部，如下面的截图所示。

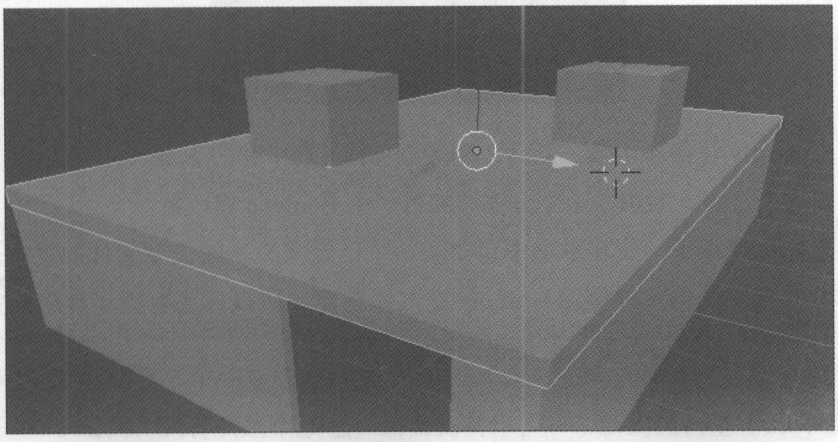


天窗是使用另一个立方体制作成天花板并在其上面剪出来的洞。

1. 用 Shift＋A 添加一个立方体，把它缩放至天窗大小，将它移动到你需要开窗的位置上。

2. 摆放立方体的 z 轴，让它能穿过天花板。

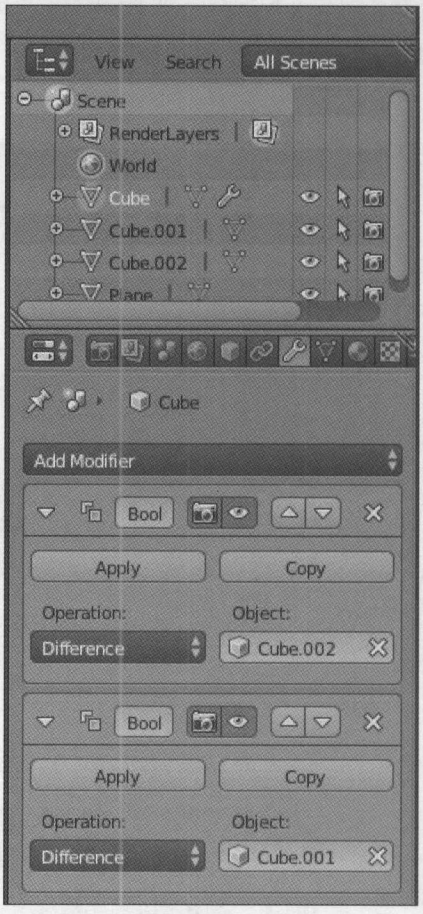
3. 用 Shift＋D 复制这个立方体，再把它移动到另一个天窗的位置上，如下图所示：



4. 用右键选择天花板。

5. 在最右边的 **Properties Editor** 面板中选择扳手图标。

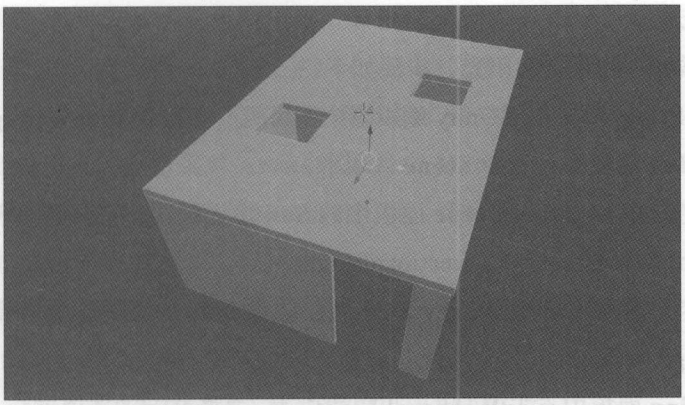
6. 然后，点击菜单 **Add Modifier | Boolean**，再对 **Operation** 选项选择 **Difference**。对于 Object 选项，选择第一个立方体（Cube.001）：



7. 点击 **Apply** 让操作生效。然后，删除立方体（选中并按 X 键）。

8. 重复这个过程，为第二个立方体添加另一个 Boolean 修改器。

9. 如果你搞迷糊了，我在本书中包含了一个已经完成了的模型文件的复本。作为替代，这个模型足够简单，以便用 Unity 的立方体构建。所以，还有很多理应被完成的东西使其成为更逼真的架构模型，但是我们将维持原样：



8.2　用 Unity 组装场景

现在，我们可以在 Unity 中使用画廊模型并添加一块地板和一个带天窗的天花板。我们将对墙应用纹理并添加光照。

8.2.1　画廊

首先，我们通过下面的步骤构建画廊：

1. 通过菜单 **File | New Scene** 创建一个新场景。

2. 通过菜单 **GameObject | 3D Object | Plane** 创建一个地板平面，重置其 **Transform** 选项并重命名为 Floor。

3. 为地板创建材质并给它着色成米黄色。

4. 我们的房间的大小是 7.3m×11m。一个 Unity 的平面是 10 个单位的方块，所以把 **Scale** 设置成（0.73, 1, 1.1）。

5. 导入画廊模型（比如 Gallery.blend），从 **Project Assets** 拖动一个复本到 **Scene** 中，重置其 **Transform** 选项。

6. 按需要手动旋转或缩放它，让它适合地板（我的适合，但是它的 **Rotate Y** 值需要设置成 90）。如果你第一次把 **Scene** 视图变成 **Top Iso** 可能有用。

7. 最好添加一个碰撞器到墙上，这样角色就不会穿过墙体了。点击菜单 **Add Component | Physics | Mesh Collider** 来实现这一步。

注意，当我们导入在 Blender 中定义的模型时，**Gallery** 中有用于各面墙和天花板的单独的对象。会创建一个带有中性灰的 **Abledo**（204, 204, 204）的材质（名称可能是 unnamed），对于墙面，我喜欢这个颜色，然而我制作了一个新的纯白色（255, 255, 255）

的材质用于天花板。

接下来，添加一些天空和阳光，步骤如下：

1. 如果 **Lighting** 选项卡在 Unity 编辑器中不可见，点击 **Window | Lighting**。

2. 在 **Lighting** 面板中，选择 **Scene** 选项卡。

3. 对于阳光，在 **Lighting Scene** 面板中的 **Sun** 输入框内，选择最右侧的圆形图标以打开 **Select Light** 对话框并选择 Directional Light。

4. 对于天空，你可以使用 Wispy Skybox，就像我们在第 7 章中所做的一样。导入 WispySky.package 文件。

5. 在 **Lighting** 面板中点击 **Window | Lighting**，选择 **Scene** 选项卡。

6. 在 **Skybox** 字段中，点击最右边的圆形图标以打开 **Select Material** 对话框并选择名为 WispySkyboxMat 的材质。

8.2.2 艺术品部件

现在，我们可以设计这个艺术展了。毫无疑问，你很熟悉如何在传统网站上使用相册，浏览一组图片集或像动态幻灯片一样查看它们。这部分代码，相框 div 布局以及 CSS 样式表一起定义，图片列表单独定义，我们将实现一些类似的内容。

首先，我们将用一个相框、光照和位置创建一个艺术品部件。然后，我们将这个艺术品挂在相册的墙上。之后，我们使用真实的图片。艺术品部件将包含一个相框（一个立方体）、一个图片平面（一个四边形）和一个点光源，所有与艺术品相关的东西都放在墙上。我们在 **Scene** 视图中创建第一个并保存成一个预制件，再把整个相册复制到墙上。我建议在 **Scene** 视图中操作这些事情。让我们开始吧：

1. 点击菜单 **GameObject | Create Empty** 创建一个容器对象，命名为 ArtworkRig。

2. 创建相框。选中 ArtworkRig，点击右键并点击 **GameObject | 3D Object | Cube**，命名为 ArtFrame。在 **Inspector** 中，设置其 **Scale Z** 为 0.05，另外，让我们假设一个 3:4 的宽高比。所以，把 **Scale Y** 的值设置成 0.75。

3. 把 rig 放在一面墙上（建筑设计图中面对入口右上角的那面墙）。可以帮你隐藏 Gallery 对象的天花板子对象（反选其 **Enable** 复选框）。然后，用 **Scene** 面板右上角的 **Scene View** 小部件把 **Scene** 视图变成 **Top and Iso**。在 **Top** 视图中点击绿色的 **Y** 图标，而在 **Iso** 视图中点击中间的方块图标。

4. 选中 ArtworkRig，确保 **Translate** 工具是激活状态（面板左边的图标栏的第二个

图标），然后使用 x 轴和 z 轴箭头进行定位。确认选择并移动 ArtworkRig。把相框的位置设置成（0, 0, 0），把高度保持在视平线（Y=1.4）的位置。适合我的 **Transform Position** 的值是（2, 1.4, –1.82），**Rotation** 是（0, 0, 0），如步骤 7 所示。

5. 使相框变成黑色。点击菜单 **Assets | Create | Material**，命名为 FrameMaterial，并设置其 Albedo 颜色为黑色。然后在 **Hierarchy** 中，选择 **Frame** 选项并在 **Inspector** 中把 FrameMaterial 材质拖进它里面。

6. 制作图片占位符。在 **Hierarchy** 中选择 ArtFrame，点击右键并点击菜单 **3D Object | Quad**，命名为 Image。把它放在相框的前方使其可见。设置其 **Position** 值为（0, 0, –0.03）并通过把 **Scale** 设置成（0.9, 0.65, 1）将其缩放得比相框略小。

7. 为了更好地领会当前的比例和视平线，试着插入一个 **Ethan** 的复本到场景中：

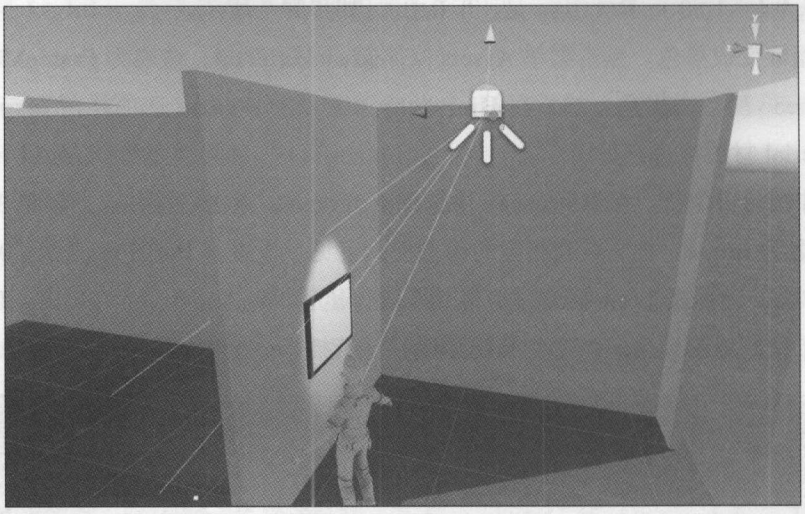
接下来，我们要在这个小部件上添加一个点光源，步骤如下：

1. 首先，通过对 **Gallery** 的子对象勾选 **Enable** 复选框选项把天花板放回原处。

2. 在 **Hierarchy** 中选择 ArtworkRig，点击右键，点击菜单 **Light | Spotlight**，然后把它放在离墙 1m 之外的地方（Z=–1.5）且上方离天花板近一些。准确的高度并不重要，因为我们并不是真的有一个照明设备。对于光源我们只有一个 Vector 3 位置值，我把 **Position** 设置成（1, 2, –1.5）。

3. 现在，调整 Spotlight 的值让它刚好照到艺术品上。我把 **Rotation X** 设置成 51，

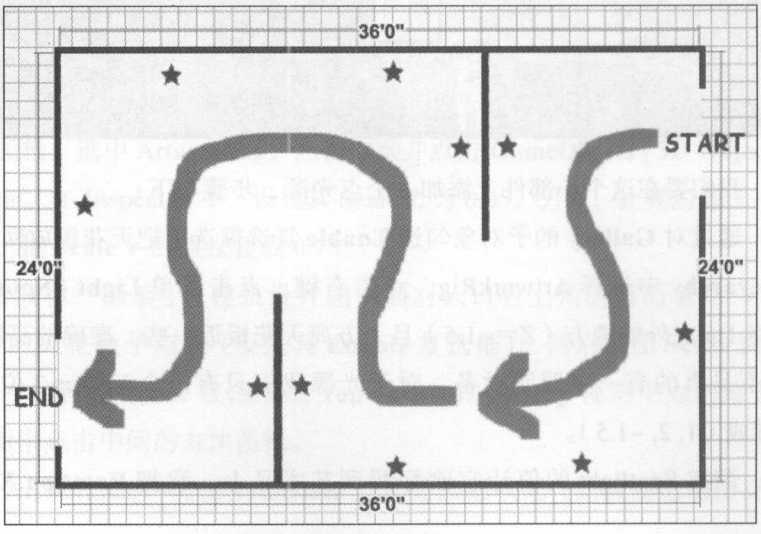
Spot Angle 设置成 30，**Intensity** 设置成 6，以及 **Range** 设置成 4。结果展示在下图中。另外，如果你看见光柱由于摄像机的角度不同而变暗，就把 **Render Mode** 设置成 Important：



4. 要想把这个小部件保存成 Prefab，在 **Hierarchy** 选择 ArtworkRig 并拖进 Project Assets 文件夹。

8.2.3 展览计划

下一步请在每一面你想要显示图片的墙上复制 ArtworkRig，按需要改变它的位置和旋转值。如果你按照下图计划来操作，你的展览将显示 10 张图片，图中用星星予以表示：



下面是在每一面墙上复制 ArtworkRig 的步骤：

1. 像之前一样，很容易隐藏天花板并把 **Scene View** 面板变成 **Top and Iso**。

2. 在 **Scene View** 面板的左上方，改变 **Transform Gizmo Toggles** 使工具项放在 **Pivot** 点而不是 **Center**。

针对每一个位置，在画廊中放置一幅艺术品，步骤如下：

1. 在 **Hierarchy** 中选择一个现有的 ArtworkRig。

2. 在 **Duplicate** 上点击右键或按 Ctrl＋D 键复制 ArtworkRig。

3. 通过设置 **Rotation Y** 为 0、90、180 或 -90，旋转这个部件让它面对正确的方向。

4. 在墙上放置这个部件。

我的画廊具体设置如下表所示：

	Position X	Position Z	Rotation Y
0	2	-1.8	0
1	-1.25	-5.28	-180
2	-3.45	-3.5	-90
3	-3.45	0	-90
4	-2	1.6	0
5	2	-1.7	180
6	3.5	0	90
7	3.5	3.5	90
8	1.25	5.15	0
9	-2	1.7	180

注意，对象在列表中的顺序我们将使用于场景中的漫游动画。在 **Hierarchy** 中放置它们，同 Artworks 的子对象一样。

8.3　添加图片到画廊中

请在你的相册中找 10 张你喜欢的照片并把它们添加到一个新的名为 Photos 的 **Project Assets** 文件夹中。按照下面的步骤添加照片到画廊中：

1. 创建照片文件夹，点击菜单 **Assets | Create | Folder**，命名为 Photos。

2. 从 **File Explorer** 中拖拽 10 张照片导入到你刚才创建的 Photos 文件夹（或点击 **Assets | Import New Asset...**）。

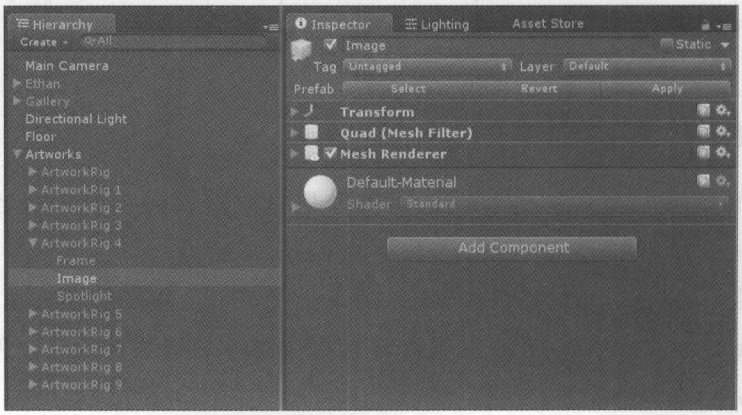
3. 现在，我们要写一段脚本用于放置 **Artworks Images**。在 **Hierarchy** 中，选择 Artworks，然后在 **Inspector** 中，点击 **Add Component | New Script** 并命名为 PopulateArtFrames。

4. 在 **MonoDevelop** 中打开这个新的脚本。

编辑 PopulateArtFrames.cs 的代码，如下（我已经写了比基本需求要多的代码以便进行解释）：

```
using UnityEngine;
using System.Collections;
public class PopulateArtFrames : MonoBehaviour {
  public Texture[] images;
  void Start () {
    int imageIndex = 0;
    foreach (Transform artwork in transform) {
      GameObject art = artwork.FindChild("Image").gameObject;
      Renderer rend = art.GetComponent<Renderer>();
      Shader shader = Shader.Find("Standard");
      Material mat = new Material( shader );
      mat.mainTexture = images[imageIndex];
      rend.material = mat;
      imageIndex++;
      if (imageIndex == images.Length) imageIndex = 0;
    }
  }
}
```

这段脚本是如何运行的？你知道的，就像下面这张截图，每个 ArtworkRig 包含一个子图片四边形（quad）。我们将为每个 quad 创建一个新的材质（替换其 **Default-Material**）并为它指定一张特殊的纹理图片。



在脚本中，我们做的第一件事是声明一个 **Texture** 的数组，命名为 Images。我们使用 Unity 编辑器中的实际图片赋值给这个参数。然后，我们定义一个 `Start()` 函数，将这些图片放于一个 ArtworkRig。

`foreach(Transform artwork in transform)` 这一行循环代码通过遍历每一个 **Artworks** 的子 ArtworkRig 赋值给 `artwork` 循环变量。

对于每一个 `artwork`，我们找到其子图片 quad 对象，赋值给名为 `art` 的本地变量，然后取得其 `Renderer` 组件（赋值给 `rend`）。

接下来，我们创建一个名为 `mat` 的新的材质，用于 `Standard Shader`。它接收一张来自图片数组的图片纹理（`mat.mainTexture = images[imageIndex]`）。

我们增加 `imageIndex` 的值，让每一个 ArtworkRig 按顺序从图片数组中取得不相同的图片。注意，如果 ArtworkRigs 比图片的数量多，它将从头开始取照片而不是导致错误。

要实现这些，让我们完成以下这些步骤：

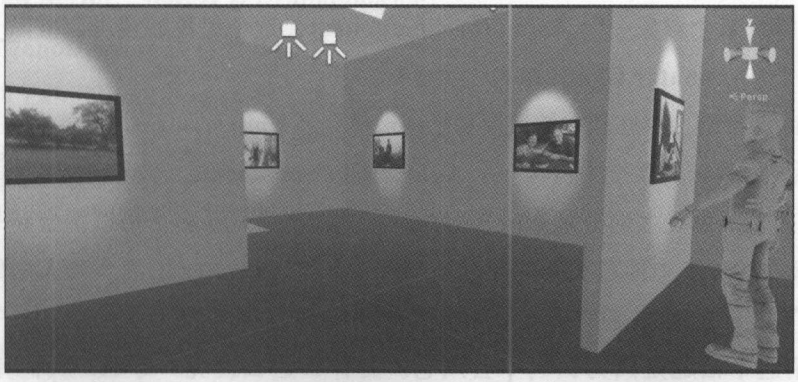
1. 保存脚本并返回到 Unity 编辑器。

2. 在 **Hierarchy** 中选择 Artworks，在 **Inspector** 中展开 **Populate Art Frames** 脚本组件，再展开 **Images** 参数。

3. 将 **Image Size** 值设置成 10。

4. 在 Project / Assets / 目录下的 Photos 文件夹中找到你要导入的图片并依次拖动它们到 **Element 0** 至 **Element 9** 的 **Images** 槽中。

当你点击 **Play mode** 时，场景中的插画将按照你指定的顺序被填充在图片中：



额外挑战：由于我们用一个单独的数组来管理图片列表，所以图片可以来自任何地方。我们手动导入它们到 Unity 中来构建这个数组，但是你可以扩展这个程

序以便从外部来获取图片，比如你的硬盘、网站、Facebook 相册或者摄像机的相册。

8.4 漫游动画

接下来，我们要实现场景的一段漫游动画。在传统的游戏中，它被用于影片剪辑，也就是一个组合起来的漫游动画像是从一关过渡到另一关。但在 VR 中，会有些不同。漫游可以真正在 VR 中得以体验。头盔跟踪始终呈激活状态，所以并不是简单的预先录制的视频。你可以向四周看并体验它们，更像是一个游乐园。这通常被称为轨道式（on-the-rails）VR 体验。

8.4.1 Unity 的动画系统

Unity 的动画系统是一套很强大的多部件系统，可以让你构建高级和错综复杂的动画行为。这套动画系统有一些不同而又听起来相似的部分：

- **Animation Curve** 为特定的属性定义一根样条曲线，这个属性的值会随着时间而变化。
- **Animation Clip** 是一个命名曲线组，可以用于某个对象并播放（它能够让这个对象产生动画效果），并且可以在 **Animation View** 面板中编辑。
- **Animator Controller** 保留那些应该被播放和当片断应该被修改或混合在一起时的片断的轨迹，可以用 **Animator** 面板的可视化编辑器来编辑它。
- 可以给某个对象一个 **Animator Component**，这个组件后续可以被赋予一个（或多个）**Animator Controller**。

在 Unity 5 之前，存在独立的遗留和已经被整合进 Unity 的单一主动画系统的 Mecanim 人型动画系统。遗留的系统中包含一个 **Animation Component**，已经不再使用了。

Unity 的动画系统大部分是面向人型动画，或者至少是从多个可以整合到一起且能够响应其他事件的片断中构建复杂的动画。**Animator Controller** 的细节和人型角色的移动动画超出了本书的范围，但它实在令人神魂颠倒。

我介绍这个话题是因为我们不会用它。为什么这么说呢？好吧，我们用于这个项目的动画只是一个简单的 **Transform** 更新。那么，这里使用动画系统就像是杀鸡用了牛刀，但其来龙去脉是很重要的。给对象一个 **Animation Component**，此组件引用编

排一个或多个 **Animation Clips** 的 **Animator Controller**。**Animation Clips** 包含你想要添加动画属性的关键帧曲线。我们将不用其他东西制作，仅使用自己的 **Animation Curves**。

8.4.2　脚本动画

对于本项目，我们将添加一个脚本到第一人称角色（MeMyselfEye）上以便随着时间来改变其 **Transform** 的位置和旋转值。它通过定义关键帧的变换而运行。就像名字所说的，对于关键帧动画，我们在指定的关键时间点定义摄像机的位置，中间的帧会自动被计算出来。

我们的第一个任务是决定关键结点放在轨迹的哪个地方，以及摄像机应该面向哪边。我们将对第一人称角色设置三个参数：位置 X、位置 Z 和旋转 Y。你可以用内置的 **Animation View** 面板手动完成，然而我们要用一段脚本代替手动方式。

在我们的画廊旅程中，我想要确认游客可以看见墙上的每一张图片。所以，对于每个关键帧，当用户浏览展览时应该直接位于艺术品的面前。要想很好地实现，需要确认 **Artwroks** 下的 ArtworkRigs 在你要浏览的正确位置上。我们将每个关键位置放在距离相框中间 1m 的地方，使用下面的步骤：

1. 如果有必要，可以从 Prefabs 文件夹 Project Assets / 目录下把 MeMyselfEye 拖进 **Hierarchy** 放在入口处（比如 $X=2.6$，$Z=-4.7$），再删除默认的 **Main Camera** 选项。

2. 在 **Hierarchy** 中，选择 MeMyselfEye。然后，在 **Inspector** 中点击菜单 **Add Component | New Script**，并命名为 ExhibitionRide。

3. 在 **MonoDevelop** 中打开新脚本。

4. 编辑 ExhibitionRide.cs 脚本，如下：

```
using UnityEngine;
using System.Collections;
public class ExhibitionRide : MonoBehaviour {
  public GameObject artworks;
  public float startDelay = 3f;
  public float transitionTime = 5f;
private AnimationCurve xCurve, zCurve, rCurve;
void Start () {
  int count = artworks.transform.childCount + 1;
  Keyframe[] xKeys = new Keyframe[count];
  Keyframe[] zKeys = new Keyframe[count];
  Keyframe[] rKeys = new Keyframe[count];
```

```
        int i = 0;
        float time = startDelay;
        xKeys [0] = new Keyframe (time, transform.position.x);
        zKeys [0] = new Keyframe (time, transform.position.z);
        rKeys [0] = new Keyframe (time, transform.rotation.y);
        foreach (Transform artwork in artworks.transform) {
          i++;
          time += transitionTime;
          Vector3 pos = artwork.position - artwork.forward;
          xKeys[i] = new Keyframe( time, pos.x );
          zKeys[i] = new Keyframe( time, pos.z );
          rKeys[i] = new Keyframe( time, artwork.rotation.y );
        }
        xCurve = new AnimationCurve (xKeys);
        zCurve = new AnimationCurve (zKeys);
        rCurve = new AnimationCurve (rKeys);
      }
      void Update () {
        transform.position = new Vector3 (xCurve.Evaluate
    (Time.time), transform.position.y, zCurve.Evaluate
    (Time.time));
        Quaternion rot = transform.rotation;
        rot.y = rCurve.Evaluate (Time.time);
        transform.rotation = rot;
      }
    }
```

在这段脚本中，我们声明了3个公有参数。GameObject artworks变量应该被初始化成其父类（Artworks）的对象，其中包含ArtworkRig对象，它们将在动画中按顺序被看到。startDelay变量是浏览开始前的第二个数字，而transitionTime变量是从一个浏览点（关键帧）到另一个浏览点所花的时间。

我们还定义了3个私有的AnimationCurve变量，名为xCurve、zCurve和rCurve，用于每个观察点的X和Z位置以及y轴上的旋转。Start()函数初始化这3条曲线。

AnimationCurve对象被定义成1个Keyframes数组。我们将让用于每个观察点的1个关键帧加上初始位置。因此，关键帧数组的第一个元素被设置到初始的MeMyselfEye transform.position.x、transform.position.z和transform.rotation.y值上。对于后面每个关键帧，我们使用艺术品的**Transform**来找到空间中的1个相框前1m的点（Vector3 pos = artwork.position - artwork.forward）。

然后，在Update()中，我们更新transform.position和transform.rotation

属性到插值，用于当前的 `Time.time` 属性。

保存脚本后执行下面的步骤：

1. 在 **Hierarchy** 中，选择 MeMyselfEye。

2. 把 Artworks 对象（包含所有的 ArtworkRigs）拖到 **Exhibition Ride（Script）Artworks** 参数栏里。

当你试运行场景时，你会开启一段美妙的画廊漫游，在每张图片前会有短暂的停留。很棒吧！很可能，你会拿起图片展示给你所有的朋友和家人！

在这段脚本中，我直接使用了 `transform.rotation` 四元数的 y 值。通常不建议直接操作 Quaternion 的值，但是因为我们只改变 1 个轴的值，所以是安全的。

额外挑战：你可以添加很多东西来加强和完善这段 VR 体验。添加 1 个引导屏和 1 个“点击开始”的按钮。添加声音到 ArtworkRig 中让它播放恰当的声音或一段解释每个艺术品细节的叙述。你可以修改图片和 / 或相框的大小以适应各种需求。你还可以调整观察距离、浏览每张图片的速度，等等。就像之前提到的，你可以在运行时从外部加载图片，比如网页、手机或 Facebook！

8.5　优化性能和舒适感

应当承认，本章大部分是关于场景设计而非特意关于虚拟现实。然而，我并不想只给你们一些预先准备的 Unity 数据包，然后说：“嗨，导入这些数据包我们就可以漫游了！”

截至现在我们应当非常清楚，VR 开发包含很多方面（一语双关）。当你使用模型、纹理和光照创建了一个“高级”场景并为你的访客提供一次惊险刺激的穿越后，你将不可避免地开始思考关于渲染的性能、帧率、延迟及晕动症。

除了最明显的渲染错误之外，我们开发者快速地变得对所有问题免疫，这样的结果是当我们测试自己的代码时我们才是那个最糟糕的人。这为开发者引出了一种新颖的、令人兴奋的辩解说辞，“在我的机器上运行良好呀”——在这里变成了“在我的大脑中运行良好呀”。

——Tom Forsyth，Oculus

专家们写了有很多非常棒的文章，有些概括了惯用的用于优化 VR 体验游戏的策略和技术。撰写本书之时，这里有一些不错的参考：

- *Squeezing Performance out of Your Unity Gear VR Game*, by Chris Pruett, Oculus, May 12, 2015, https://developer.oculus.com/blog/squeezing-performance-out-of-your-unity-gear-vr-game/
- *12 Performance Tricks for Optimizing VR Apps in Unity*, by Darshan Shankar, VR developer, May 12, 2015, http://dshankar.svbtle.com/performance-optimization-for-vr-apps
- *Unity VR Optimisation Hints and Tips*, Nick Pittom, VR developer "Crystal Rift", October 31, 2014, http://gamasutra.com/blogs/Nick-Pittom/ 20141031/229074/Unity_VR_optimisation_hints_and_tips.php
- *Optimizing Graphics Performance*, Unity Manual, http://docs.unity3d.com/460/Documentation/Manual/OptimizingGraphicsPerformance.html
- *Practical Guide to Optimization for Mobiles*, Unity Manual, http://docs.unity3d.com/Manual/MobileOptimizationPracticalGuide.html

这些课程和技术中的许多内容既可以用于移动游戏开发又可以用于 VR。

针对 VR 做开发是一个移动目标——平台硬件、SDK 软件以及 Unity 3D 引擎自身都变化和改进得相当快。随着产品的改进和新开发者产生的新的感悟，博客和网页文章被迅速地更新或替代。就像本书的其他部分，我需要问自己我们是否能够把这些分解成基本的即使细节改变也将保持不变的原则。

总之，下面列出一些专注于设计的优化技术和每一项的局限之处：

- 你的游戏的实现和内容。
- Unity 引擎的渲染流水线。
- 目标平台的硬件和驱动程序。

接下来的讨论是对你所能做之事的调研，以及相关延伸阅读和研究的简介。

8.5.1 优化实现和内容

你拥有最多控制权的一样东西是你构建的内容。有些影响性能的决策是有意识的、有目的的和创新的。也许你想要带有高保真声音的超现实图形，因为它非常棒！变化也许会构成一个很难的设计方案。然而，它很可能也会带来一点小创新的思考和实验，可以达到（或接近）完全相同的视觉结果且性能更好。质量不仅是外观，还包括感受。所以，优化用户体验是一个基本决策。

8.5.1.1 简化模型

最小化模型网格中的顶点和面的数量，避免复杂的网格。移除你永远不会看见

的面，比如固体内部的面。清除重复的顶点并移除复本。很可能你创建模型的程序中就有这样的工具，比如 Blender 就有这样的工具。另外，你还可以购买第三方的工具以简化模型网格。对于动态批处理（看下一节）需要小于 300 个（非静态）顶点的网格。

8.5.1.2　使用纹理贴图代替复杂的网格

如果你不介意多边形的数量，你可以使用含有很多个顶点的复杂网格为表面细节和纹理凸起来建模。当然，这在计算上是不明智的。用纹理贴图来救援！考虑用法线贴图对比高度贴图。

法线贴图是用于伪造模型表面凸起和凹陷光照的简单纹理贴图。在某些 3D 渲染中，你可以相当成功地使用法线贴图；但在 VR 中，这样就看起来太平了。当你移动头部时，它看起来就像是墙纸。

高度贴图则使用一张纹理图片来制作一种非常传统的 3D 表面几何图形。高度贴图比法线贴图优越是因为它们不仅定义了表面凸起和凹陷，而且提供了平行视差。然而，作为一个着色器，它们的计算开销很大，只是没有用网格的开销那么大。参考 http://docs.unity3d.com/Manual/StandardShaderMaterialParameters.html。

8.5.1.3　限制需要绘制的对象

遮挡剔除（occlusion culling），在摄像机看不见对象时禁用对它们的渲染，因为它们被其他对象遮挡了。请阅读 Unity 的文章，http://docs.unity3d.com/Manual/OcclusionCulling.html，学习如何在项目中设置遮挡剔除。

另一种减少场景中细节渲染的方式是使用 Global Fog，其基于距离。比迷雾限制更远的对象将不会被绘制。参考项目的图形设置《Project Graphics Settings》，http://docs.unity3d.com/Manual/class-GraphicsSettings.html。

对细节分级或 LOD 分组，是一种简化几何对象的好方法，近处的物体用细节模型渲染，而那些远处的模型则用简化的模型渲染。Unity 可以在摄像机靠近时自动在每个 LOD 几何对象间切换，参考 http://docs.unity3d.com/Manual/class-LODGroup.html。

你拥有最多控制权的一样东西是你构建的内容。最小化你的模型网格中的顶点和面的数量。高度贴图对于复杂几何图形可能是一种有效的替代。你尝试减少需要被渲染对象的数量越多，效果越好。

8.5.1.4 光照和阴影的性能

你还有一种很好的控制光照数量、光照类型及它们的摆放和设置的处理方法。在任何可能的时候使用烘焙的光照贴图，它能预计算光照效果到一个单独的图片中，而不是在运行时计算。

节约使用实时阴影，当某个对象投射阴影时会生成一个阴影贴图，它会被用于渲染其他可能接收阴影的对象。阴影有很高的渲染开销而且通常需要高端的 GPU 硬件。阅读 Unity 手册中的文章，可以在链接 http://docs.unity3d.com/Manual/LightPerformance.html 找到。

其他技术，比如灯光探测器（实时或烘焙）和着色器的选择（以及着色器选项），可以让你的场景看起来非常棒。然而，它们还能在性能上有显著的效果。在审美和图形性能间进行权衡是一门艺术和科学，我希望能在某天驾驭它。

8.5.1.5 优化脚本

你写的每个 Update( ) 回调函数都会在每一帧被调用。移除不用的更新，使用一个状态变量（和一个 if 语句）在它们不需要时停止计算。

在某种情况下，你可能得使用一个评测工具来看看你的代码的性能如何。当你的游戏在 **Play Mode** 下，Unity 的 **Profiler**（参考下一节）将运行并提供具体哪里在消耗时间以及持续时间。如果 **Profiler** 指示有大量时间花费在你写的脚本上，那么你应该考虑另一种方式来重构代码让它更加高效。通常，这与内存管理有关，但也可能与数学或物理有关，参考 http://docs.unity3d.com/Manual/MobileOptimizationPracticalScriptingOptimizations.html。

8.5.2 优化 Unity 渲染流水线

还有很多针对 Unity 如何执行其渲染的重要的性能需要考虑。其中的某些性能可能对于任何图形引擎都适用。或者，它们会在 Unity 自己的新版本中有所改变，因为渲染是一种竞争性的业务，算法可能被替代，实现可能会被重新加工。

一个来自 Unity 的 50 分钟时长的描述——Unity 渲染流水线（2014 年 1 月）可以在链接 https://unity3d.com/learn/resources/unity-rendering-pipeline 找到。

8.5.2.1 批量处理

也许，Unity 中最物有所值的功能是将不同的网格归类到一个单独的批处理中，这

个批处理会被立即放进图形硬件。这比单独发送网格快很多。网格实际上先被编译进一个 OpenGL 顶点缓存对象或一个 VBO（关于更多信息请察看 http://en.wikipedia.org/wiki/Vertex_Buffer_Object），但那是渲染流水线的低层细节。

每一个批处理调用一次绘制，在一个场景中减少调用绘制的次数比减少顶点或三角形的实际数量的效果更有意义。

共有两种类型的批处理——静态批处理和动态批处理。

首先，确认在 **Player Settings** 中启用 **Static Batching** 和 **Dynamic Batching**。

对于静态批处理，简单地通过在 Unity 的 **Inspector** 中为场景内的每个对象勾选 **Static** 复选框以标记对象为静态。把一个对象标记为静态的是告诉它将永远不能移动、动画或缩放。Unity 将自动把这些共享相同材质网格放在一起形成一个大网格。

需要说明的是，共享相同材质的网格。所有这样的网格在一个批处理中必须有相同的材质设置——相同的纹理、着色器、着色器参数以及材质的指针对象。

怎么会这样？它们是不同的对象！这可以通过把多个纹理组合成一个单独的大纹理文件或 TextureAtlas，然后 UV 映射成尽可能多的模型就好。很像是一个用于 2D 和网页图形的 sprite 图片。听起来很难？有一些第三方工具，比如 Pro Draw Call Optimize，可以帮助你完成构建。

当用脚本管理纹理时，使用 Renderer.shareMaterial 而不是 Renderer.material 以避免创建重复的材质。对象接收一个重复的材质将退出这个批处理。

你可以回忆一下本章之前的 **Gallery** 项目，我们在一段脚本中为每个 ArtworkRig 图片创建了单独的材质。如果将其标记为 **Static**，它们将不会被放进批处理。然而结果是运行良好，因为纹理是全分辨率的摄影图片——因为太大而不能与其他纹理以任何方式融合。问题在于，有时需要去担心批处理，而有时不用。

把一个对象标记为 Static 是告诉 Unity 这个对象将永远不会移动、动画或缩放。Unity 将自动地把这些把相同材质共享进一个单独的大 VBO 的网格放在一起成为一组，它之后会被立即放进图形硬件。

动态批处理与静态批处理类似。对于那些没有标记为 **Static** 的对象，Unity 将尝试把它们放进批处理，即使它会是一个更慢的过程，因为它需要考虑逐帧动画（CPU 开销）。共享材质的需求依然存在，还有其他的限制，比如顶点个数（小于 300 个顶点）和统一的 **Transform Scale** 规则，参见 http://docs.unity3d.com/Manual/DrawCallBatching.html。

当前，只有 **Mesh Renderers** 和 **Particle Systems** 使用批处理，这意味着蒙皮网格、衣服、尾迹渲染以及其他一些类型的渲染组件并没有使用批处理。

8.5.2.2 多通道像素填充

渲染流水线中的另一个关注点有时指的是像素填充率。如果你思考一下，渲染的终级目标是用正确的颜色值填充显示设备上的每个像素。如果绘制某个像素绘制的次数越多，开销会越大。举个例子，观察透明的粒子效果，比如烟雾，主要是用透明四边形连接起来的像素。实际上对于 VR，Unity 把大于物理显示维度的像素绘制进一个帧缓冲区内存中，这些像素后续进行视觉变形纠正（ocular distortion correction）——桶形效果（barrel effect）和色差纠正（chromatic aberration correction），色彩分离的后期处理，然后再折腾到 HMD 显示器上。其实，在后期处理之前可能已有多个叠加缓冲区组合到一起。

多通道像素填充就是某些高级渲染器的工作方式。光照和材质效果，比如多光照、**动态阴影以及透明度**（Transparent 和 Fade Render 模式）都是以这种方式实现的。Unity 5 的 Specular Shader 也一样。基本上都是干货！

带有材质的 VBO 批处理需要多通道像素填充提交多次，这样可以增加绘制单元的网格数目。针对于你的项目，你可以选择优化并避免通道像素填充在一起，或者理解清楚什么样的场景需要高性能，什么样的场景需要高保真，你需要仔细地策划这个场景。

针对项目，你可以选择优化并避免通道像素填充在一起，或者理解清楚什么样的场景需要高性能，什么样的场景需要高保真，你需要仔细地策划这个场景。

如果你使用烘焙光照贴图预计算光照和阴影，多通道是可以避免的。这是 Unity 4.x 的方式，使用 **Legacy Shaders/Lightmapped/Diffuse**（参见 http://docs.unity3d.com/Manual/LightmapParameters.html）。然而，如果烘焙的话，你将丢失 Unity 5 软件中很棒的新的基于物理的着色器（http://blogs.unity3d.com/2015/02/18/working-with-physi-cally-basedshading-a-practical-approach/）。

但是，你可以使用 **Light Probes** 以很低的成本模拟动态对象的动态光照。如 Unity 的手册所述：“**Light Probes** 被烘焙成立方体贴图，存储了直接的、间接的，甚至在场景中各个点位发射光源的信息。当一个动态对象移动时，它内插附近光探针的采样以逼近特定位置上的光照。这是一种用于模拟动态对象上逼真光照的廉价方式，而不需要使用

代价很高的实时灯光。”参考 http://docs.unity3d.com/Manual/LightProbes.html 获取更多信息。

在 **Quality Settings** 中把同时发生的光照的全部数量设置为 1。Oculus 的 Chris Pruett 解释：“最近的光照会被逐个像素渲染，而周围的灯光会通过球谐函数（spherical harmonics）被计算出来。”作为替代，你甚至可以放弃所有的像素光照而依赖于 Light Probes。

8.5.2.3　其他渲染技巧

Nick Pittom 建议你应该创建 2 048 分辨率的纹理并导入到默认的 1 024 的设置。这样可以加速渲染。

当讨论 GearVR 时另一个来自 Darshan Shankar 的技巧是，当你针对无阴影使用高质量的设置渲染到 Android 时，你需要切换目标平台到 PC，使用高分辨率烘焙光照并开启硬阴影和软阴影，再切换回 Android。更多信息请参见 http://dshankar.svbtle.com/developing-crossplatform-vr-apps-for-oculus-rift-and-gearvr。

你能发现或建议什么技巧吗？也许应该有人建立一个论坛之类的社区！更多信息请参见 http://forum.unity3d.com/。

8.5.3　优化目标硬件和驱动

硬件架构正朝着有利于虚拟现实（增强现实）图形流水线的性能演进。对于传统的视频游戏 VR 引入的需求不那么重要。举个例子，延迟和掉帧（渲染一帧比刷新所花费的时间更长）相比高保真 triple-AAA 的渲染功能处于次要地位。VR 需要及时渲染每一帧并做两次——每只眼睛一次。受这个新兴行业的需求所驱动，半导体和硬件厂商正在构建新型改进的架构，将不可避免地影响内容开发者思考如何优化。

也就是说，你应该为你想要发布程序的低端机器或设备开发和优化。如果这样的优化需要不合适的妥协，可以考虑分为高端和低端版本。VR 设备厂商已经着手发布最小化 / 建议的硬件规格，这些规格实现了大部分的猜测。

学习 Unity 中针对每个目标平台的 **Quality** 设置（通过点击菜单 **Edit | Project Settings | Quality**）。以设备厂商（比如 Oculus）的建议作为建议值开始，并根据需要调整。举个例子，你可以调整比如 **Pixel Light Count**、**Anti Aliasing** 和 **Soft Particles** 的参数。

比如，对于移动 VR，建议你对比 CPU 密集型和 GPU 密集型进行优化。有的游戏会

让CPU更努力地工作，其他的会影响GPU。通常，你应该偏爱CPU超过GPU。Oculus Mobile SDK（GearVR）有一个API可以用于CPU和GPU节流以便控制热量和电量的消耗。

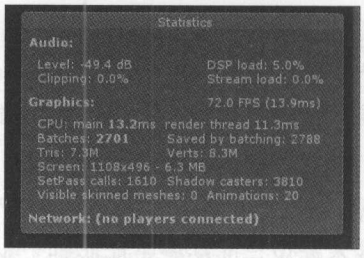
8.5.4 Unity Profiler

优化可能需要你进行大量的工作，很努力地学习也是必要的，所以需要你去认真对待。好消息是可以逐步地完成。解决得越明显，越有所值，尤其是那些很少或不会引起视觉效果减弱的优化。

Unity编辑器中包括两个内置的工具以评估性能——**Stats**窗格和**Profiler**窗格。

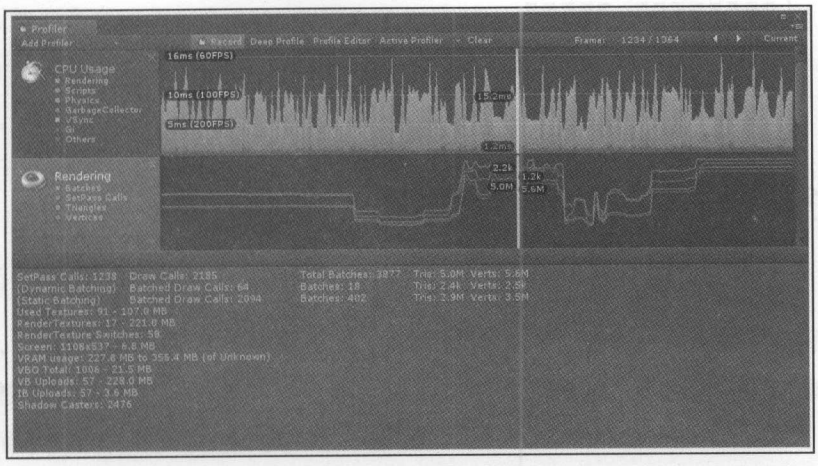
当评测和优化时，写下（或截屏）统计并标记它们，可以在一个表格中，让你能够保留过程日志并衡量尝试的每项技术的效应。

在**Game**面板中，你可以开启**Stats**窗格，如下面的截图所示，显示（除其他之外）了批处理、三角形、顶点的数量以及运行时帧率。（请参见http://docs.unity3d.com/Manual/RenderingStatistics.html）：



Unity的**Profiler**选项是一个性能探测工具，可以报告你的游戏中的各个区域花费的时长，包括渲染和脚本。它记录游戏中随着时间的统计数据并以时间线图表展示出来。点击可以逐级查看细节，参见http://docs.unity3d.com/Manual/Profiler.html。下面的截图是一个经典的Oculus Tuscany试例场景的评测：

你在开始优化前，着手处理场景中各种摄像机位置的性能，用于各种动画和动作序列以及不同的场景。写下（或截屏）统计并标记它们，可以在一个表格中，让你能够保留过程日志并衡量尝试的每项技术的效应。



小结

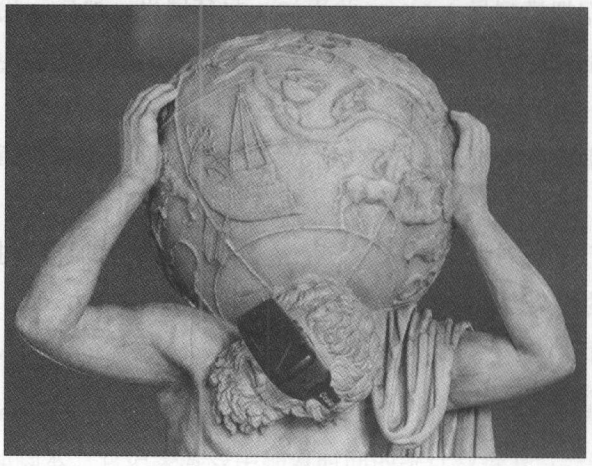
本章中，我们从零开始构建了一个画廊场景，以一个平面图开始进入 Blender 建造一个总体结构。我们把模型导入 Unity 并添加一些环境光。然后，我们构建一个艺术品部件，包含一张图片四边形、一个相框和一个点光源，之后在画廊的各个墙上放置部件。接下来，我们导入一组个人相片编写一个脚本在运行时填充艺术框。最后，我们写了一个脚本让第一人称摄像机漫游这个场景，并在每张图片面前短暂停留，实现一个简单的 VR-on-rails 体验。

然后，我们进行了针对性能和 VR 舒适度关于优化项目的详细讨论，意味着你不得不注意内容和模型的复杂度、渲染流水线，以及目标平台的能力。

在下一章中，我们将看一下使用预录制的 360° 多媒体的不同 VR 体验。你将构建和学习照片球、等距柱状投影和信息图。

Chapter 9

第 9 章
利用 360°



阿特拉斯托举着天球，2nd Cent BCE（源自 http://ancientrome.ru/art/artworken/img.htm?id=4657）

360° 的照片和视频是使用虚拟现实的不同方式，无论是体验它们还是生产及发布它们，在当今都是容易被消费者接受的。查看预先录制的图片所需要的计算量比渲染完整的 3D 场景所需的计算量少很多，而且在移动 VR 设备上运行良好。本章中，我们将探索以下话题：

- 为地球仪、全景图和照片球使用纹理。
- 理解什么是 360° 的多媒体。

- 添加一个光球和一个 360° 的视频到你的 Unity 项目中。
- 什么是视野以及如何在视野内录制一段 360° 的视频。

注意，本章中的项目都是独立的，且不直接被本书中的其他章节所依赖。如果你决定跳过其中的某些章节或者不保存，也没有关系。

9.1　360° 的多媒体

近来，术语“360°”和“虚拟现实”被大量地滥用，通常放在同一个句子中。消费者可能会被误导认为它们是相同的东西，认为它们已经弄明白了，认为它们制作起来非常容易。而事实上，并不是那么简单。

一般来说，术语“360°”指的是以一种浏览预先录制的照片或视频的方式，这种方式允许你旋转观看方向以便能够看到你视野之外的内容。

非虚拟现实的 360° 多媒体已经变得相当普遍。比如，很多房地产网站提供了一个基于网页的播放器以实现全景漫游功能，让你能够交互式地旋转视角以观察所有空间。类似地，YouTube 支持上传和播放 360° 的视频，并且提供一个带有交互控制的播放器以便在播放过程中观看各个方向。Google 地图允许你上传 360° 的静态**照片球**（photosphere）图片，更像是它们的街景工具，让你可以用一个 Android 或 iOS 应用亦或一个消费级照相机创建这样的图片（更多信息请参考 https://www.google.com/maps/about/contribute/photosphere/）。互联网上有很多 360° 的多媒体！

使用 VR 头盔观看 360° 多媒体具有惊人的沉浸感，即使是静态的图片。你正站在用一张图片映射到其内表面上的球体的中心，但是你却感觉像是真的在那个（用摄像机）拍摄到的场景之中。简单地转动你的头部向四周看，这正是当人们第一次看见 VR 时令人对其产生兴趣的东西之一，并且它在 Google Cardboard 和 GearVR 上也是一个流行的应用。

本章中，我们将探索利用 Unity 和 VR 的功能使用所有 360° 的多媒体的各种用途。

9.2　水晶球

作为开始，让我们先娱乐一下，我们使用一张规则的矩形图片作为纹理应用到球体上，看看它做了什么以及它看起来有多糟糕。

首先，通过下面的步骤创建一个新的场景用于本章：

1. 通过菜单 **File | New Scene** 创建一个新的场景。然后，点击 **File | Save Scene As...** 并命名为 360Degrees。

2. 通过菜单 **GameObject | 3D Object | Plane** 创建一个平面，并使用 **Transform** 组件的齿轮图标 | **Reset** 重置其变换值。

3. 从 project 的预制件中插入一个 MeMyselfEye 的实例，设置其 **Position** 值为（0, 1, –1），然后删除默认的 Main Camera。

现在，在创建第一个球体的时候编写一个旋转脚本。我使用本书中提供的图片 EthanSkull.png（拖进 **Project Assets/Textures** 文件夹）。然后，执行下面的步骤：

1. 通过菜单 **GameObject | 3D Object | Sphere** 创建一个新的球体，使用 **Transform** 组件的齿轮图标 | **Reset** 重置其变换值。

2. 设置其 **Position** 值为（0, 1.5, 0）。

3. 把命为 EthanSkull 的纹理拖到球体上（你可以使用任何你想用的照片）。

4. 通过菜单 **Add Component | New Script** 创建一个新的脚本并命名为 Rotator。

打开 rotator.cs 脚本并编辑，如下：

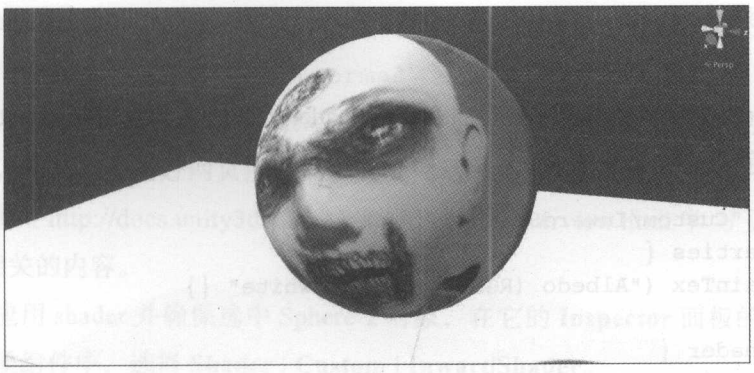
```
using UnityEngine;
using System.Collections;

public class Rotator : MonoBehaviour {
  public float xRate = 0f; // degrees per second
  public float yRate = 0f;
  public float zRate = 0f;

  void Update () {
    transform.Rotate (new Vector3 (xRate, yRate, zRate)*
  Time.deltaTime);
  }
}
```

然后，设置旋转率使它绕着 y 轴以 20° 每秒的速度旋转，步骤如下：

1. 在 **Rotator Script** 组件上，设置 **Rates** 的 ***X***，***Y***，***Z*** 为（0, 20, 0）。

2. 保存场景并在 VR 中试验。

看起来吓人吧？别怕。映射上去的图片可能被扭曲了，但是它看起来很酷。对于有些应用，些许的扭曲是有艺术意图的，你不用害怕它。



额外挑战：尝试在 Unity 5 中用基于物理的着色器和反射探头让球体看起来更像水晶玻璃（更多信息请查看 http://blogs.unity3d.com/2014/10/29/physically-based-shading-in-unity-5-a-primer/）。

9.3　魔法球

下一个例子，我们将从内部观看球体，把一张普通图片映射到球内表面上。然后，我们会把一个坚硬的有颜色的壳放在它的外表，所以你需要实际地走到球体上并把头伸进去看看里面都有什么！

参考下面的步骤来制作它（我使用我们在第 2 章中引入的图片 GrandCanyon.png，但是你也可以使用任何图片，最好是一张带有开阔天空的风景图）：

1. 将 Sphere 1 移开，并把它的 **Position** 设置成（–3, 1.5, –1.5）。

2. 通过菜单 **GameObject | 3D Object | Sphere** 创建一个新的球体，把它的 **Position** 设置成（0, 1.5, 0），并命名为 Sphere 2。

3. 通过反选复选框禁用其 **Sphere Collider** 组件。

4. 通过菜单 **Assets | Create | Material** 创建一种新的材质，并命名为 GrandCanyon-Inward。

5. 把 GrandCanyonInward 材质拖到 Sphere 2 上。

6. 找到 GrandCanyon 纹理图片并把它放到 GrandCanyonInward 组件的 **Albedo** 纹理上（球体最左边的 **Albedo** 字段）。

现在，我们有一个带有图片封装了表面的球体，与前一个例子相同。我们将翻转它让纹理渲染在其内表面。我们用一个自定义的着色器来实现，步骤如下：

1. 通过菜单 **Assets | Create | Shader** 创建一个新的自定义的着色器，并命名为 InwardShader。

2. 在 **Project Assets** 中，在新建的 InwardShader 上双击使其在 MonoDevelop 中打开。编辑着色器的定义文件如下：

```
Shader "Custom/InwardShader" {
  Properties {
    _MainTex ("Albedo (RGB)", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType"="Opaque" }
    LOD 200

    Cull Front

    CGPROGRAM
    #pragma surface surf Standard vertex:vert

    void vert(inout appdata_full v) {
      v.normal.xyz = v.normal * -1;
    }

    sampler2D _MainTex;

    struct Input {
      float2 uv_MainTex;
    };

    void surf (Input IN, inout SurfaceOutputStandard o) {
      fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
      o.Albedo = c.rgb;
    }
    ENDCG
  }
  FallBack "Diffuse"
}
```

Unity shader 是一个文本脚本文件，包含一些 Unity 渲染流水线动作的指令。通读上述这段 shader 代码，我们自定义的 shader 命名为 `InwardShader`。它有一个 **Properties** 面板——**Albedo** 纹理可以在 Unity 的编辑器中设置。我们把 `Cull` 模式设置成 `Front`（可选的值分别是 `Back`、`Front` 和 `Off`）；单词 **cull** 在这里的意思是“需要

被移除或忽略”。这样，它将忽略前方的表面，而渲染后方的表面。我们还要翻转每个面的法向量（`v.normal.xyz = v.normal * -1`）用于光照计算。`surf()` 采样函数由 Unity 的渲染器调用，应用 shader 中定义的属性，在本例中是仅有的主要纹理图片。

实话说，我其实是通过网页搜索发现这段代码的，研究并简化它使之用于本例。你可以通过链接 http://docs.unity3d.com/Manual/ShadersOverview.html 学习更多 Unity 的 ShaderLab 相关的内容。

现在，应用 shader 并确保选中 Sphere 2 对象，在它的 **Inspector** 面板的 GrandCanyonInsider 材质组件中，选择 **Shader | Custom | InwardShader**。

在 VR 中试运行。从外表面，它看起来特别怪，但是走进球体向四周看。哦？看起来很棒。

对于移动，你可能需要把 HeadLookWalk 脚本（在第 6 章中编写的）添加到 MeMyselfEye，或者就使用 Unity 标准的 FPSController 预制件替代 MeMyselfEye。

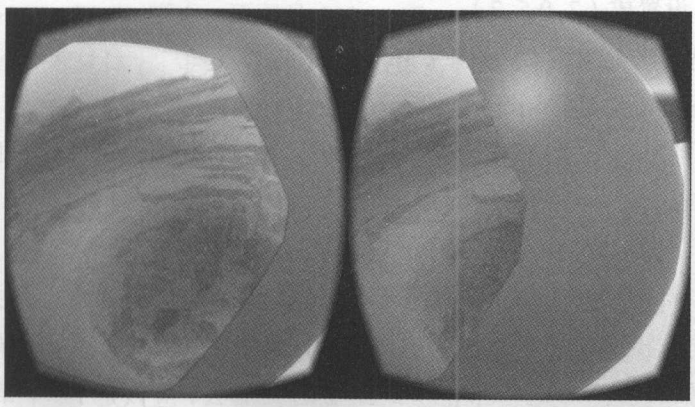
最后，我们要用一个固体有色圆球把它包围起来，步骤如下：

1. 在 **Hierarchy** 中选中 Sphere 2，右击选择 **3D Object | Sphere** 使新的球体成为 Sphere 2 的子对象。

2. 通过反选禁用它的 **Sphere Collider** 组件。

3. 找一个固体材质，比如我们上一章制作的名为 Red 的材质，并把它拖到新的球体上。

在 VR 中试运行。下图是我所看到的截图，就像在凝视一个蛋壳！



我们创建了两个同心球体——同样的大小且同样的位置。我们禁用它们的碰撞器，这样我们就可以穿透其表面。第一个球体有一个默认的着色器，它渲染朝外的表面（红

色的），另一个球体使用自定义的名为 InwardShader 的着色器。

作为第一人称角色，你可以进入球体并向四周看。如果你使用支持位置跟踪的头盔，比如 Oculus Rift，那么你不需要实际地走进球体。你可以停在它前面然后前倾让你的头部进入它！像是在变魔术吧！

9.4 全景图

假如图片很宽，比如你用手机拍的全景照片，会有什么不同？让我们试一下并把它映射到球面上。

我去年在洛杉矶徒步登上了好莱坞的地标的顶部，在那里我用手机拍了两张 180° 的全景图片，后来我快速地用 Gimp 把图片拼接了起来。Hollywood.png 图片已经包含在本书中，你也可以用你自己的全景图片：

把它们放在一起，步骤如下：

1. 把 Sphere 2 移开，并把它的 **Position** 设置成（–3, 1.5, 0）。

2. 通过菜单 **GameObject | 3D Object | Cylinder** 创建一个新的圆柱体，把它的 **Position** 设置成（0, 1.5, 0），并命名为 Cylinder 3。

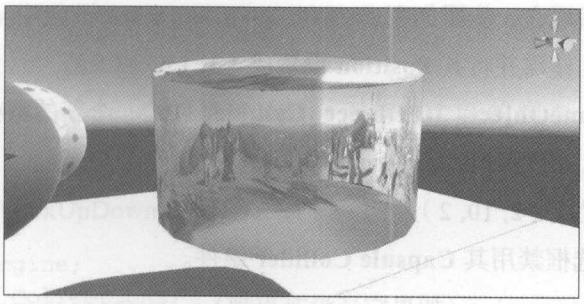
3. 把 **Scale** 设置成（2, 0.5, 2）。

4. 通过反选复选框禁用 **Capsule Collider** 组件。

5. 把 Hollywood 纹理拖到圆柱体上。

6. 确保选中 Cylinder 3 对象。在 **Inspector** 面板中的 Hollywood 材质组件上，点击菜单 **Shader | Custom | InwardShader**。

Unity 中默认的圆柱体是 2 个单位高度以及 1 个单位宽度（半径为 0.5 个单位）。我的图片的分辨率是 5 242×780，宽高比大约是 6.75∶1。应该用多大的圆柱体来遵循这个宽高比且不让图片变形呢？如果你把 Y 设置成 0.5 来归一化高度到 1 个单位，那么我们需要 6.75 个单位的周长。让 circumference$=2\pi r$，那么 r 应该是 1.0 个单位。所以，我们把 **Transform** 组件的 **Scale** 设置成（2, 0.5, 2）。那么，结果大概是这样。



在 VR 中试运行，走进圆柱体并向四周看。

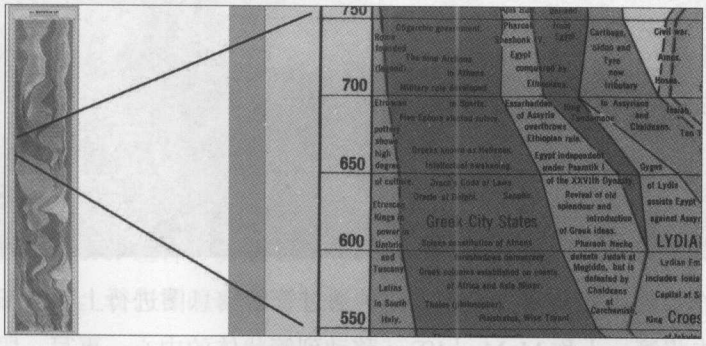
令人不舒服的是，纹理图片也被映射到了圆柱体的两头（顶部平面和底部平面上）。如果你有兴趣，可以用 Blender 制作一个两头不带面的合适的圆柱体。

然而，全景图以“Year 2009”这种方式还是很有趣的，真正的答案是全景圆柱体对于 VR 来说还相当糟糕。不过，在 VR 中还有些其他的圆柱投影的应用程序，尤其对于显示二维定量的信息，我们接下来将会探讨。

9.5　信息图

信息图是一个可视化图表，或者用于表达信息或数据的图。要找范例的话，可以用 Google 图片搜索引擎搜索“infographics”。信息图可以非常大，以致于在台式计算机屏幕上很难察看，更不用说手机了。但是，我们在 VR 中可以使用几乎无限的空间，我们来看看。

我无意中发现了这张漂亮的描述世界历史的历史地图，产于 1931 年，由 John B.Sparks 为 Rand McNally 所绘。原图是 158cm 高 ×31cm 宽，宽高比大约为 1∶5。我们将把它映射到半个圆柱体上，因为是 180°，所以需要两次：



历史地图，由 John B.Sparks，Rand McNally 在 1931 年制作（参考源：http://www.davidrumsey.com/luna/ servlet/detail/RUMSEY~8~1~200375~3001080:The-Histomap-）。

让我们用 VR 实现它，步骤如下：

1. 将 Cylinder 3 移出并将其 **Position** 设置成（3, 1.5, –3.5）。

2. 通过菜单 **GameObject | 3D Object | Cylinder** 创建一个新的圆柱体，将其 **Position** 设置成（0, 11.5, 0），然后命名为 Cylinder 4。

3. 把 **Scale** 设置成（2, 10, 2）。

4. 通过反选复选框禁用其 **Capsule Collider** 组件。

5. 导入图片 Histomap.jpg（如果它还没有出现）。

6. 在 **Project Assets** 中选择 Histomap 纹理，在 **Inspector** 面板上的 **Import** 设置中，把 **Max Size** 设置成最大值 8 192，这样文本将会保持可读。然后，点击 **Apply**。

7. 把 Histomap 纹理拖到圆柱体上。

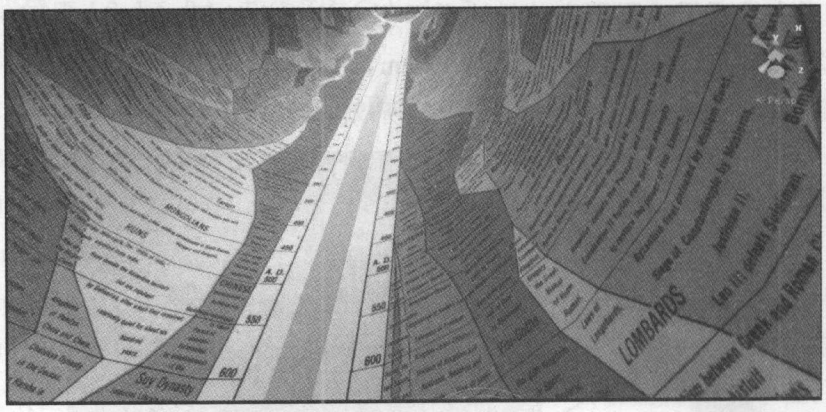
8. 在 **Inspector** 上选中 Cylinder 4，在 **Histomap** 材质组件上设置 **Tiling X**＝–1。

9. 在 **Histomap** 材质组件上，点击菜单 **Shader | Custom | InwardShader**。

就像前一个例子一样，我们可以使用图表的宽高比来计算正确的缩放比，从而解释了我们为什么平铺两次并且默认半径是 0.5 个单位的原因，**Scale** 值为（2, 10, 2）时表现很好。

我们需要可以得到的最大像素分辨率为 8 192，让文本具有可读性。我们还需要通过将 X-Tiling 设置成负值从内部反转这个可读文本的纹理。

下面的图片是管状信息图的内部，向顶部看：



要计算这个投影，我们需要给用户一种通过管状信息图进行上移和下移的方式。简单起见，我们先将第一人称 MeMyselfEye 移动到管状体的中心，再写一段抬头向上看的脚本，如下：

1. 在 **Hierarchy** 中选择 MeMyselfEye，然后通过选择 **Transform** 组件的齿轮图标 | **Reset** 重置其变换值。

2. 通过菜单 **Add Component** | **New Script** 创建一个新的脚本，命名为 HeadLookUpDown。

编辑脚本 HeadLookUpDown.cs 脚本，如下：

```
using UnityEngine;
using System.Collections;

public class HeadLookUpDown : MonoBehaviour {
  public float velocity = 0.7f;
  public float maxHeight = 20f;

  void Update () {
    float moveY = Camera.main.transform.forward.y * velocity *
    Time.deltaTime;
    float newY = transform.position.y + moveY;
    if (newY >= 0f && newY < maxHeight) {
      transform.position = new Vector3 (transform.position.x,
      newY, transform.position.z);
    }
  }
}
```

这段脚本使用你注视的方向（`Camera.main.transform.forward`）确定垂直偏移量，这个值用于当前变换的位置值（最小值与最大值的差值）。

在 VR 中试运行。现在非常有趣了！

要想进一步加强，我们可以添加一个 **Spotlight** 作为 MeMyselfEye 的 Head 对象的子对象。这样当你移动头部时，它将照亮你正在阅读的内容，就像是矿工的头灯。

1. 在 **Hierarchy** 中，找到 MeMyselfEye 的子对象 Main Camera。

2. 右键点击 **Light** | **Spotlight**。

3. 调整 **Light** 参数为，**Range**：4，**Spot Angle**：140，**Intensity**：0.6。

4. 要想有一个更引人注目的效果，如果有来自管状外部的外围光照和影子干扰，你可能需要在 **Mesh Renderer** 组件中反选 **Receive Shadows** 复选框和禁用 **Directional Light**。

在 VR 中试运行。我们有阅读灯了！

9.6 等距圆柱投影

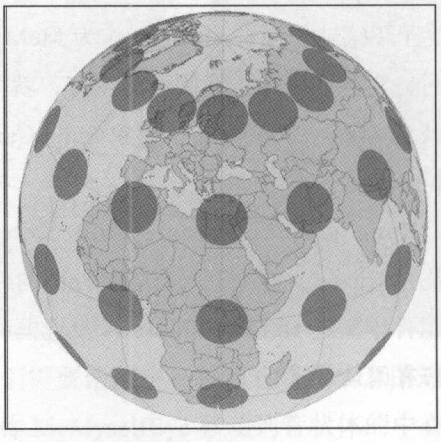
自从地球被发现是圆的，地图制作者和航海员就已经争论过如何把球形的地球投影到二维图上。投影的种类很多而且历史让人神魂颠倒（如果你被这种事难住了），结果是地球的某些区域的变形是必然的。

如果想学习更多有关地图投影和球体变形的知识，请参考 http://en.wikipedia.org/wiki/Map_projection。

作为计算机图形设计师，与老水手相比可能对投影不觉得那么神秘，是因为我们知道 UV 纹理映射。

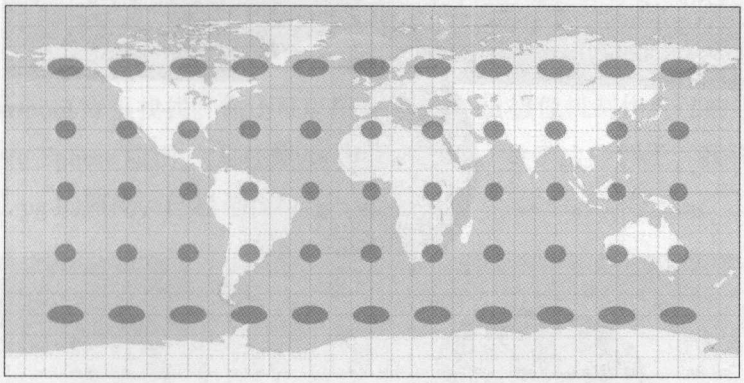
3D 计算机模型在 Unity 中通过网格定义为：一组用边连接起来的 Vector 3 的点集组成三角形的面。你可以把一个网格展开（比如用 Blender）成一个平的 2D 构造，用来把纹理像素定义到网格表面的相应区域上（UV 坐标）。一个地球仪，当被展开时会产生变形，就像展开的网格所定义的那样。结果图被称为 UV 纹理影像。

在计算机图形建模时，UV 映射可以是随意的，并且取决于美术需求。但是对于 360° 多媒体，典型的是用**等距圆柱投影**（或子午线投影）完成（更多信息请参考 http://en.wikipedia.org/wiki/Equirectangular_projection），球体被拆成一个柱体投影，随着你向南北极推进而拉伸纹理，并保持经线均为等距的竖向直线。下图为 Tissot's Indicatrix（更多信息请参考 http://en.wikipedia.org/wiki/Tissot%27s_indicatrix），展示了一个地球，上面巧妙地排列了多个完全相同的圆形（插图来自 Stefan Kühn）：



Tissot's Indicatrix 插图，Stefan Kühn, Creative Commons license（源自 https://en.wiki-pedia.org/wiki/Tissot%27s_indicatrix）

下图显示了一个用等距柱状投影展开的地球。



等距柱状投影插图，Eric Gaba (Sting), Wikinedia Commons Licence（源自 http://en.wikipedia.org/wiki/Equirect angular_projection）

我们将一个等距柱状的网格用于我们的图片球，并对其纹理映射使用一个合适的投影（包装）图片。

9.7　地球仪

首先，让我们使用一个标准的 Unity 球体，比如我们在之前那个带有地球纹理图片的 Crystal Ball 例子中使用的球体。导入本书中的 Tissot_euirectangular.png 图片，步骤如下：

1. 将 Cylinder 4 移出 way 并将其 **Position** 设置成（–3, 0, –3.5）。

2. 通过菜单 **GameObject | 3D Object | Sphere** 创建一个新的球体，并将其 **Position** 设置为（0, 1.5, 0），并命名为 Sphere 5。如果有需要的话可以添加 Rotator 脚本。

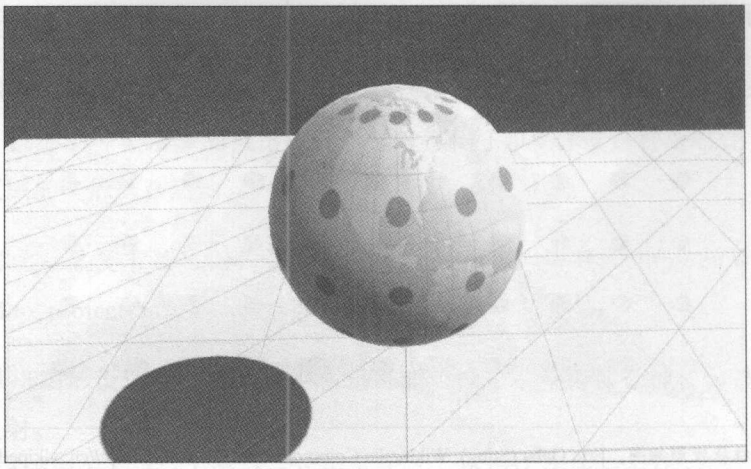
3. 把名为 Tissot_equirectangular 的纹理拖到球体上。

4. 在 VR 中试运行，看一下这个地球仪，如下图所示：

注意，令人遗憾的是，除赤道外 Tissot 的圆都是椭圆形而非圆形。原来是因为 Unity 中提供的默认的球体对等距柱状纹理贴图配合得不是很好。作为替代，我已经提供了一个特别设计：PhotoSphere.fbx（它是 3D Studio Max 中默认的球体模型）。让我们试一下：

1. 把 Sphere 5 移出并将其 **Position** 设置成（3, 1.5, 1.5）。

2. 导入文件。从 **Project** 面板中（如果你已创建一个，Assets/Models 文件可能已被选中），点击右键并选择 **Import New Asset...**，然后找到 PhotoSphere.fbx 文件并导入。

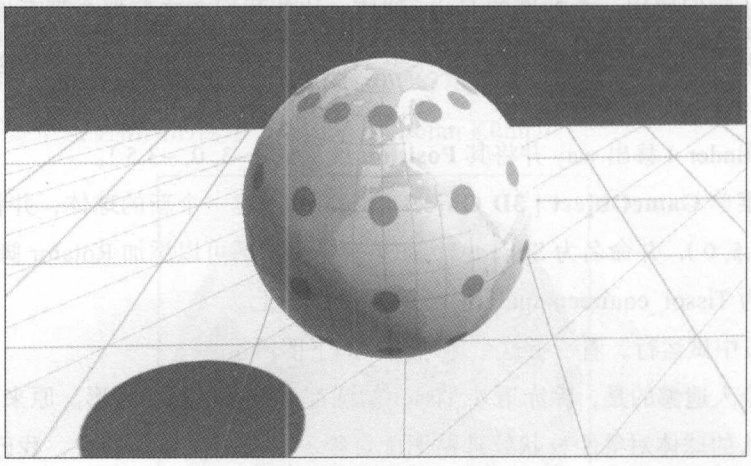


3. 通过把 PhotoSphere 预制件从 **Project Assets** 中拖拽到 **Scene** 中创建一个新的等距柱状球体。

4. 设置其 **Position** 值为（0, 1.5, 0），并命名为 Sphere 6。如果需要的话添加 Rotator 脚本。

5. 把名为 Tissot_equirectangular 的纹理拖到球体上。

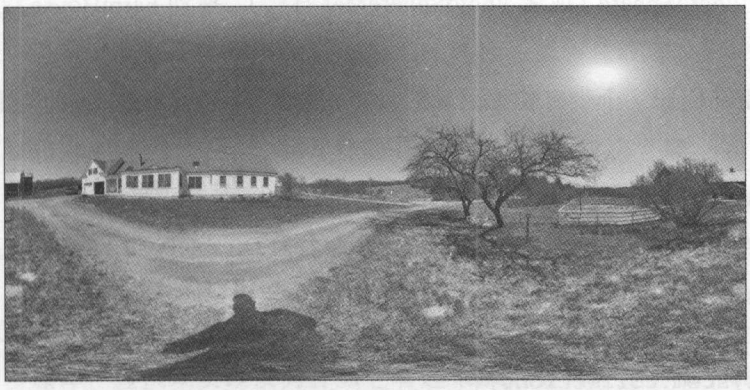
在 VR 中试运行。好多了。你现在可以看到纹理被正确地映射了，圆圈也圆了。



9.8 照片球

是的，先生，这东西现在很流行。它好过全景，好过自拍，可能甚至好过 Snapchat！我们最终于看到了你一直期待的东西！这就是 360° 照片球！

我们在本章中已经涵盖了很多话题，这些话题使得我们可以很轻松地讨论 360° 照片球。现在，我们所需要的就是一张 360° 的照片。你可以试着在 Google Images 上搜索一张 360° 的照片，或者也可以在 Flickr 的照片池（(https://www.flickr.com/groups/equirectangular/pool/) 里找找。你还可以用 Ricoh Theta 摄像机制作自己的照片，或者用 Google 的 Photo Sphere 应用，适用于 Android 和 iOS。对于这些例子，我将使用一张名为 FarmHouse.jpg 的照片，已经包含在本书中：



让我们来制作它。下面是在 Unity 中制作这样一张照片的完整步骤，以一个新的空的场景开始：

1. 通过点击菜单 **File | New Scene** 创建一个新的场景。然后，点击 **File | Save Scene** 并命名为 PhotoSphere。

2. 通过从 **Project Assets/Models** 中把 PhotoSphere 模型（在前一个例子中的 PhotoSphere.fbx 中导入的）拖进场景中，创建一个等距柱状球体。

3. 通过点击其 **Transform** 组件的齿轮图标 | **Reset** 进行重置。

4. 设置其 **Scale** 为（10, 10, 10）。

5. 通过菜单 **Assets | Create | Material** 创建一个材质，并重命名为 PhotoSphereMat。

6. 选中 PhotoSphereMat，点击 **Shader | Custom | InwardShader**（之前在章节 9.3 中创建的）。

7. 如果场景中还有其他物体，你可能需要禁用阴影。在 **Mesh Renderer** 组件中，反选 **Receive Shadows** 复选框。

8. 在场景中禁用全局的 **Directional Light**（反选之）。

9. 照亮球体内部。通过点击 **GameObject | Light | Point Light** 添加一个点光源，并

通过其 **Transform** 组件的齿轮图标 | Reset 重置。

10. 删除场景中的 **Main Camera**。

11. 从 Project Assets 中拖动一个 MeMyselfEye 的实例到场景中，并设置其 **Position** 为（0, -0.4, 0）。

12. 导入你想用的照片，我们的照片名称是 FarmHouse.jpg。在 **Import** 设置中，选择最大的 **Max Size** 为 8 192。

13. 选中 PhotoSphere（或者 PhotoSphereMat 自己），拖动 FarmHouse 纹理到 **Albedo** 纹理碎片上。

如果你正在使用一个带有位置跟踪的设备，比如 Oculus Rift，我们需要禁用它，步骤如下：

1. 在 **Hierarchy** 中选择 MemMyselfEye 对象。

2. 点击 **Add Component | New Script**，并命名为 DisablePositionalTracking。

编辑 DisablePositionalTracking.cs 脚本，如下：

```
using UnityEngine;
using System.Collections;

public class DisablePositionalTracking : MonoBehaviour {
  private Vector3 position;

  void Start () {
    position = Camera.main.transform.position;
  }

  void Update () {
    Camera.main.transform.position = position;
  }
}
```

这段脚本只是在每次 update 时把摄像机的位置重置到起始位置。保存脚本并在 VR 中试运行。

提示一下，球体、材质及自定义 shader 的创建已经涵盖在本章中前面的部分了。

MeMyselfEye 预制件在第 3 章中已经被创建，让摄像机位于其中心上方 0.4 个单位之处。我们想让摄像机处于原点（0, 0, 0）。在 VR 中正确体验照片球的方式是在绝对中心，你可以在任何方向上转动你的头部向四周看，位置跟踪应该被禁用。（另一个用于头部位置固定的技术是把 MeMyselfEye 放大到一个很大的尺寸。）

把一个点光源放在正中心也是一个好主意。虽然可能你的图片有明显的光源，但是它可以有效地定位一个或多个处于球体中其他位置的光源。

也许，还有一种改进自定义 shader 的方式，就是让它不接收外部的阴影或光源。但是，我们需要手动来做这些调整。

要想切换图片，重复最后两步——导入资源并指定给 PhotoSphereMat 材质的 **Albedo** 纹理。如果你想在游戏中进行这一步，你可能需要使用脚本，比如使用 Material.mainTexture( )。

播放 360° 视频也类似，你使用 MovieTexture 替代一张图片纹理。影片纹理在移动设备上有些限制，更多细节请阅读 Unity 的文档（http://docs.unity3d.com/Manual/class-MovieTexture.html），或者考虑使用 Unity 资源商店中的第三方解决方案。

9.9　视野

好了，我们看过了球体、圆柱体、球和魔法球，都很有意思！现在让我们聊一些更多关于 360° 多媒体、虚拟现实以及为什么它看起来那么引人注目。在平面屏幕上与在 VR 头盔内部看观看 360° 视频有巨大的不同。为什么呢？

在电影院中、计算机屏幕上或 VR 中，可视区域的一条边到另一条边的角度通常是指视角或**视野（field of view，FOV）**。举个例子，比传统电影院具有更大屏幕的 IMAX 电影院包含更多周边视觉（**peripheral vision**）以及更宽的视野。下面的表格比较了各种观看体验下的横向视野。更宽的视野对于提供沉浸感很重要。

这些 FOV 值是朝着一个方向看过去而不移动头部或眼睛：

观看体验	水平视野（FOV）
27" 计算机显示器	26°
电影院	54°
IMAX 影院	70°
Google Cardboard	90°
Oculus Rift DK2	100°
GoPro 摄像机	74° ～140°
人类视觉	180°

在 Unity 中，**Camera** 组件用于渲染一个可配置 FOV 角度的场景。这个设置项调整

视角矩形或者说视锥体（frustum）的大小用于窥视场景，就像是调整一个真实的摄像机镜头的焦距（比如，正常的与广角的进行对比）。这个视角映射到物理显示器上，成为电视、显示器或移动设备的屏幕。

对于更多关于在传统视频游戏中 FOV 调整的信息请阅读好文：《All about of View》，（July 18, 2014）http://steamcommunity.com/sharedfiles/filedetails/?id=287241027。

类似地，VR 头盔显示器的内部是一个具有实际宽高的手机大小的屏幕。然而，通过头盔的镜头，屏幕的宽度看上去要大很多（且距离很远）。

在 VR 中，你并不是那么明显地受限于视野（FOV）和屏幕的物理维度，因为你可以很容易地随时移动你的头部来改变你的观察方向。这样就提供了一个完整地沉浸视图，水平的 360°，就和你向左右上下 180° 来回看时一样。在 VR 中，视野仅对周边视觉的外围和眼球的运动有意义。

概括地说，下面是观察 360° 照片球的原理：

1. 假设我们有一个等距柱状球体，由错综复杂的小三角面组成。
2. 等距柱状投影的纹理被映射到球体内部。
3. 我们有一个 VR 摄像机架设在球体中心，并且有左眼和右眼两个摄像机。
4. 两个摄像机看向球体内部的某一块区域，由视野限定。
5. 当你在 VR 中移动头部时，摄像机所看的区域与你的移动一致。
6. 你将得到一片连续的 360° 图片的视野。

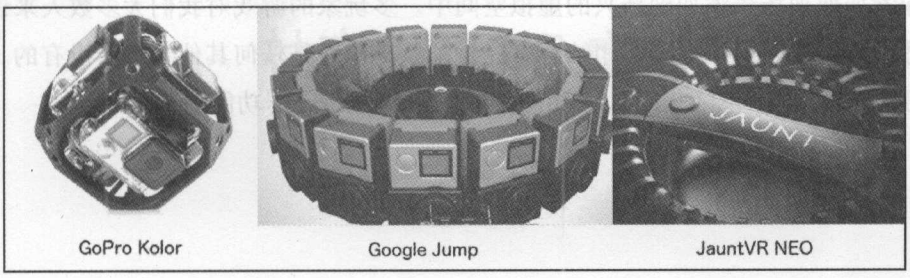
这可能不是实时的计算机图形，但是，伙伴们，这还是很棒的。

9.10 捕捉 360° 多媒体

目前为止，我们已经讨论了单视场的（monoscopic）多媒体——360° 的照片或用单个镜头录制的视频，虽然是从所有方向录制的。当在 VR 中观看时，是的，有左眼和右眼，但这是同一张映射到球体上的平面图片的立体视图。通过遮挡它并没有提供任何平行视差（real parallax）或深度幻觉（illusion of depth）。你可以转到你的头部，但你不能移动处于球体中心的位置。否则，沉浸的错觉将会破坏。

那 360° 立体画（360-degree stereo）如何呢？假设两只眼睛的照片球有偏移量的话会怎么样呢？立体的 360° 录制和播放是一个非常困难的问题。

要录制非立体的 360° 的多媒体，你可以使用 1 台像 GoPro Kolorp 这样的设备，下面最左边那张图片即是，它用 6 个单独的 GoPro 摄像机同时记录 6 个方向。同步的视频之后通过特殊的高级图片软件拼接在一起，很像要本章开头提到的 Photo Sphere 应用，而这里是视频。



要录制立体的 360° 多媒体，你需要像一台像 Google Jump 这样的设备，它在 1 个圆柱体队列排列 16 个单独的 GoPro 摄像机，上图中中间那张图即是。之后运用批量的云处理功能可以在几小时内通过高级图片处理软件构造立体视图。

更多关于像 Google Jump 这种 16 个摄像头队列运行细节的解释请阅读这篇文章：*Stereographic 3D Panoramic Images*, by Paul Bourke (May, 2002) by visiting http://paulbourke.net/stereographics/stereopanoramic/。

要录制完整的沉浸式 360° 多媒体，像 JauntVR NEO 光场相机这种新技术，上图中最右边那张图即是，正在从头开始发明中，专门用于电影的 VR 内容。

时间会告诉我们这些多媒体技术将如何开发并成为我们的创新工具箱的一部分。

想要使用游戏中的 360° 图片录制器？

你可以录制一段玩家在游戏环境中的 360° 等距柱状全景并保存并上传，日后可以用由 eVRydayVR.See 创建的 Untiy 插件观看 https://www.assetstore.unity3d.com/en/#!/content/38755。

小结

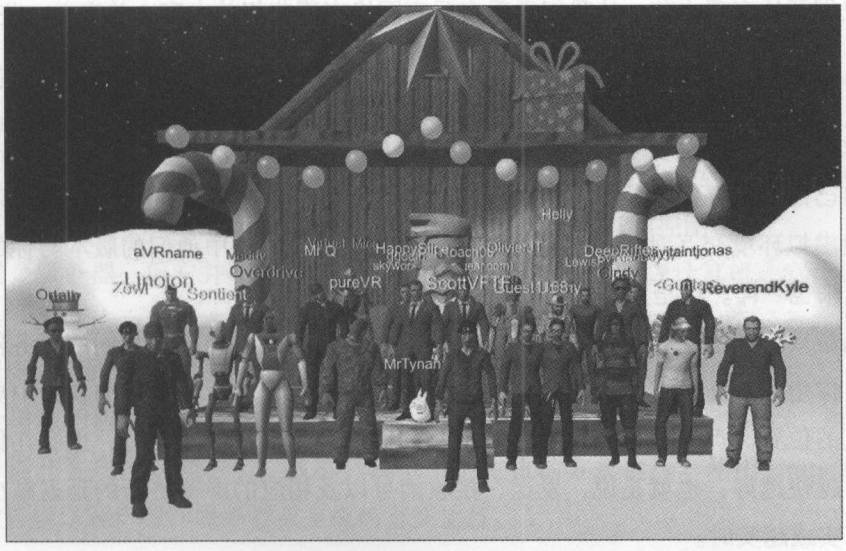
本章中，我们先通过映射一张规则图片到球体的外表面作为简单的开始，然后用一个自定义的着色器翻转它，映射这张图片到球体内部。我们探究了全景的圆柱体投影和

大型信息图。之后，我们用几个章节的内容来理解等距柱状球体投影和照片球，以及它们在虚拟现实中如何被用于360°沉浸感的多媒体。最后，我们探索了一些观看和录制360°多媒体背后的技术。

在下一章中，我们将探索社交，就像我们在Unity VR项目中添加多用户网络一样，以及探索如何把场景添加到新兴的虚拟空间中。多玩家的游戏对我们大多数人来说已经很熟悉了，但是当与虚拟现实相结合时提供了社交体验是任何其他技术所没有的。我们将学习关于网络技术和Unity 5.1新引入的Unity Networking功能。

第 10 章 *Chapter 10*

社交化的 VR 虚拟空间



那个人是我，Linojon，在前面左边戴棒球帽的那个人！重要的是，上面这张照片是 2014 年 12 月 21 日虚拟世界于圣诞前夕在一个 VRChat 直播间里拍摄的。我构建了一个季节性主题的世界，叫作 GingerLand，并邀请聊天室的好友在某个周末的聚会时来参观。然后，有的人建议说：“嘿，让我们拍一张集体照吧！”就这样，我们所有人都聚集在我那寒冷的小木屋的前廊，然后喊：“茄子！”之后，就是你们看到的这样了。

对于很多人来说，在 VR 中与其他人进行社交互动而产生的内心体验至少就像使用 Facebook 与浏览静态网页之间的不同一样，或者说就像分享 Snapchats 与浏览在线相册相比那样令人激动。它非常个性化并且有活力。如果你已经体验过，就能明白我的意思了。我们现在看看社交式 VR 体验如何用 Unity 实现。有很多途径，从打草稿到接入现有的 VR 世界。本章中，我们将讨论下面的话题：

1. 介绍多玩家的网络如何运行。
2. 使用 Unity 网络引擎实现一个在 VR 中运行的多玩家场景。
3. 构建并共享一个自定义的 VRChat 房间。

注意，本章中的项目是独立的且不直接依赖于本书中的其他章节，如果你决定跳过其中某些项目或者不保存，也无所谓。

10.1 多玩家网络

在我们开始实现之前，让我们看看多玩家网络有哪些相关内容，并定义一些术语。

10.1.1 网络服务

考虑一下你正在运行着一个连接到服务器的 VR 应用程序，而你的一些朋友也同时在他们自己的 VR 装备上运行相同的程序。当你在游戏中移动你的第一视角、射击物体或与其他虚拟环境交互时，你期望其他玩家也能看到。游戏中他们的版本与你的版本同步，反过来也一样。这是怎么做到的呢？

你的游戏创建了一个到服务器的连接，其他玩家也同时连接到相同的服务器。当你移动时，你角色的新位置广播到每一个其他连接的玩家，然后在他们的视野中更新你的虚拟角色的位置。类似地，当你的游戏接收到其他角色的位置改变时，也会在你的视野中更新。越快越好，也就是说，发送、接收消息以及相应的屏幕更新的延迟越少，交互感就越真实或越实时。

多玩家服务应该能帮助你管理所有活跃客户端之间游戏状态的共享、新玩家和物体的产生、安全因素，以及低层网络连接、协议和服务质量（比如数据的速率和性能）。

网络由一系列层级构建而成，低层处理数据传输的细节而对于数据的内容是不知道的。中间层和更高层提供越来越多的聚合功能，这些功能也可能更直接地对网络应用有

所帮助。对我们而言，在多玩家游戏和社交 VR 中，高层级理念上将用最小的自定义脚本提供所有你需要在游戏中实现的多玩家功能，而通过一个简洁的 API 访问其他层将便于你有特殊的需求。

还有一些多玩家的服务可以使用，包括 Exit Games 的 Photon、Google、Apple、Microsoft、Amazon 等的其他服务。

流行的 Photon 云服务可以很容易地添加 Unity Asset Store 中的 **Photon Unity Networking（PUN）**包（更多信息请参考 https://www.assetstore.unity3d.com/#/content/ 1786）。如果你对在 VR 中尝试 Photon 感兴趣，可以查看 http://www.convrge.co/multi-playeroculus-rift-games-in-unity-tutorial，一篇来自 Convrge 的博文。

Unity 5 有自己的内置 Unity 网络系统，最近被从头开始重写了，与 Unity 4 中的相比有很大的提升。Unity 的网络系统减少了自定义脚本的需求并提供了一个丰富的功能集组件，API 也紧密地与 Unity 相结合。

10.1.2　网络架构

网络的关键是客户端到服务器的系统架构。在当今世界中网络随处可见——你的网页浏览器是一个客户端，而网站被托管在服务器上。你喜欢的音乐收听应用是一个客户端，而它的流服务是一个服务器。类似地，每一个游戏实例在连接到网络时就是一个客户端，它与服务器对话并在所有其他游戏客户端之间传达状态和控制信息。

我说是服务器，但它并不需要是某个地方的一台单独的物理计算机。它可以是，也可以不是。最好认为客户端和服务器是进程——一个程序的实例或一个运行在某个地方的应用程序。**云服务器**是一个虚拟进程，它可以通过互联网即服务访问。

一个单独的应用某些时候可以在同一时间既作为客户端又作为服务器。对于后者，你的服务器和客户端是同一个，故称其是以一个主机的方式运行。使用 Unity 的网络系统，游戏可以以客户端、服务器或主机的方式运行。

即使如此，对于游戏实例之间通信，一个公有网络协议（**IP**）地址还是需要的。一个轻量的中继服务器可以用最少的资源提供这个服务。

10.1.3　本地与服务器

在 Unity 中，你可以使用脚本在游戏过程中创建或实例化新的对象。在多玩家的情况下，这些对象需要在本地和网络上都被激活或者说被孵化出来，这样所有的客户端就

都知道了。**出生系统**（spawning system）管理所有跨客户端的对象。

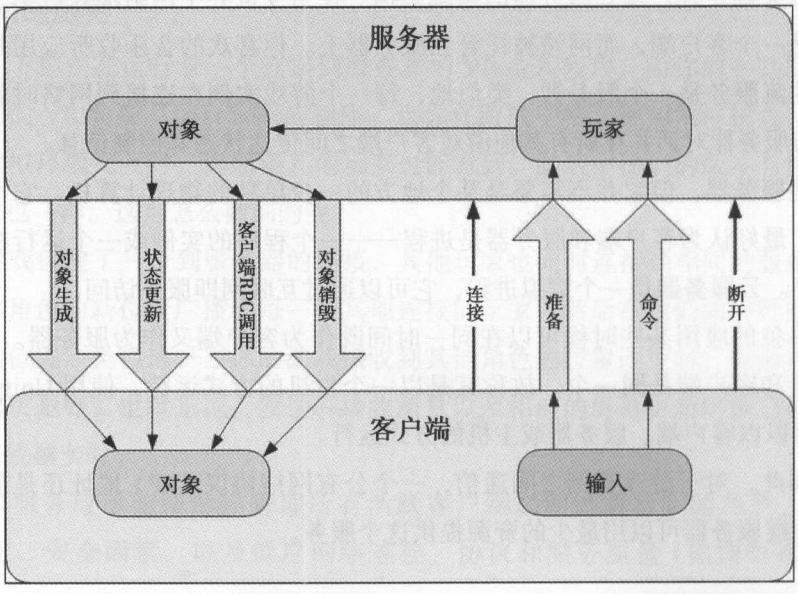
区别本地和网络玩家对象很重要。本地玩家对象在你的客户端属于你。

举个例子，在第一人称体验中，玩家的虚拟角色将会被一个摄像机组件生成，然而其他玩家的虚拟角色则不会。这也是一个重要的安全因素，以防止玩家破解游戏和改变其他玩家的角色。

本地玩家对象有本地认证，也就是说，玩家对象负责控制自己，比如自己的移动。否则，对象的创建、移动和销毁就不由玩家控制了，验证应该放在服务器端。

另一方面，服务器验证也是需要的。举个例子，当游戏在随机位置上创建敌人时，你可能想让所有的客户端都得知那个随机位置。当一个新玩家加入正进行中的游戏时，服务器将帮助创建并设置好当前游戏中的活跃对象。你不会想让一个对象出现在默认位置然后在与其他客户端同步时闪到其他位置上。

下面这张来自 Unity 文档的图片显示了通过网络执行的操作。服务器生成**远程过程调用**（RPC）让客户端创建或更新对象。



客户端发送命令到服务器以执行操作，然后命令被传达到所有远程客户端。

实时网络系统是一门深入的工程学科。分层网络架构的目标是简化理解，让你从晦涩的细节中抽身。

总结起来，就是性能、安全和稳定性。如果你需要在多玩家游戏中调试或优化，你可能需要深挖并实践以获取针对下层原理更好的理解。

10.1.4　Unity 的网络系统

Unity 的网络引擎包括一组强健的高层组件（脚本），使得添加多玩家功能到游戏中变得容易。还有些更重要的组件，包括 Network Identity、Network Behavior、Network Transform 和 Network Manager。

Network Identity 组件对于每个可能在客户端创建的游戏对象预制件都是必需的。在组件内部，它提供了一个唯一的资源 ID 和其他参数，使这个对象可以毫不含糊地被跨越网络识别和创建。

Network Behavior 类继承于 MonoBehavior 并提供了网络函数的脚本。相同细节的文档被整理在 http://docs.unity3d.com/Manual/class-NetworkBehaviour.html。

当你想要同步对象的移动和物理属性时，添加 Network Transform 组件。它就像是更为通用的 SyncVar 变量与用于平滑帧更新的附加智能插值的一个快捷键。

Network Manager 组件是黏合这些组件的胶水。它处理连接的管理、跨网络对象的创建以及配置。

当新的玩家对象被创建时，你可以在 Network Manager 组件中指定一个出生点。作为替代，你可以添加游戏对象到场景中并给它们一个 Network Start Position 组件，它可以用于出生系统。

可以出生的非玩家对象也可以在 Network Manager 的出生列表中设置。另外，Network Manager 组件可以处理场景的变化并提供调试信息。

与 Network Manager 组件相关的是匹配功能，可以通过配置来匹配玩家，让他们聚合在一起同时开始游戏——一个多玩家大厅管理器（lobby manager），玩家可以设置他们自己的状态为准备开始游戏，这是很有用的功能。

10.2　建立简单的场景

让我们立即开始制作自己的多玩家演示项目吧。

出于教学的目的，我们将以一个非常简单的带有一个标准的第一人称摄像机的场景开始着手实现网络系统。然后，我们在 VR 中进行适配。

10.2.1 创建场景环境

作为开始，我们将制作一个新的包含一个地平面和一个立方体的场景，然后我们将创建一个基础的第一人称角色。

点击菜单 **File | New Scene** 创建一个新场景，然后，点击 **File | Save Scene As...** 并命名场景为 MultiPlayer。

通过菜单 **GameObject | 3D Object | Plane** 创建一个新的平面，重命名为 GroundPlane，然后使用 **Transform** 组件的齿轮图标 | **Reset** 重置其 **Transform**。通过设置 **Scale** 为（10, 1, 10）让平面变大。执行下面的步骤：

1. 用灰色材质让 GroundPlane 更好看。点击 **Assets | Create | Material**，命名为 Gray，点击它的 **Albedo** 颜色卡选择一个中性灰，比如 RGB（150, 150, 150）。把这个灰色材质拖动到 GroundPlane 上。

2. 为了提供一些背景和朝向，我们将只添加一个立方体。点击 **GameObject | 3D Object | Cube**，使用 **Transform** 组件的齿轮图标 | **Reset** 重置其 **Transform**，把 **Position** 设置靠边一些，类似（–2, 0.75, 1）这样的值。

3. 给立方体着色，点击 **Assets | Create | Material** 并命名为 Red，点击其 **Albedo** 颜色卡选择一种好看的红，比如 RGB（240, 115, 115）。把这个红色材质拖动到立方体上。

接下来，我们将添加一个标准的第一人称角色，FPSController，如下：

1. 如果你没有加载**标准 Characters** 资源包，点击菜单 **Assets | Import Package | Characters**，再选择 **Import**。

2. 在 **Project Assets / Standard Assets / Characters / FirstPersonCharacter / Prefabs / FPSController** 中找到 FPSController，并拖进场景中。

3. 用 **Transform** 组件的齿轮图标 | **Reset** 重置其 **Transform**，使它看着物体的前方，设置 **Position** 为（5, 1.4, 3），设置 **Rotation** 为（0, 225, 0）。

4. 在 **Inspector** 中选择 FPSController，在 **First Person Controller** 组件上勾选 **Is Walking** 复选框，并设置 **Walk Speed** 为 1。

如果你正在使用一个在 **Player Settings** 中开启了 VR 的项目，请暂时禁用它：

1. 点击 **File | Build Settings...**，在 **Build Settings** 对话框中选择 **Player Settings...**。然后，在 Other Settings 下方反选 **Virtual Reality Supported** 复选框。

2. 保存场景。点击 Play 模式并验证你是否可以像平常一样使用键盘（WASD）和鼠标进行移动及向四周看。

10.2.2　创建虚拟角色的头部

接下来，你将需要用一些虚拟角色来代表你自己或你的朋友。我还是要让它超级简单以便我们可以关注于原理，所以暂时不考虑整体，只制作一个浮动的带有脸的头部。下面是我的步骤。因人而异，只要确认它是面向前方的（*Z* 轴正方向）即可：

1. 创建一个虚拟角色容器。点击菜单 **GameObject | Create Empty**，重命名为 Avatar，用 **Transform** 组件的齿轮图标 | **Reset** 重置其 **Transform**，设置其 **Position** 至眼睛的高度，比如（0, 1.4, 0）。

2. 在 Avatar 结点下为头部创建一个球体。点击 **GameObject | 3D Object | Sphere**，重命名为 Head，用 **Transform** 组件的齿轮图标 | **Reset** 重置其 **Transform**，并设置 **Scale** 为（0.5, 0.5, 0.5）。

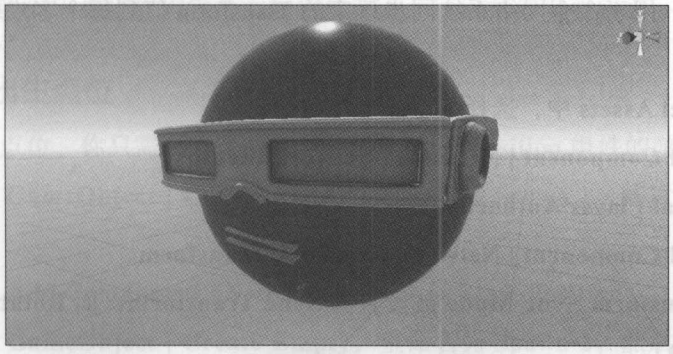
3. 给头部着色。点击 **Assets | Create | Material** 并命名为 Blue，点击其 **Albedo** 颜色卡选择一种好看的蓝色，比如 RGB（115, 115, 240），拖动这个 Blue 材质到 Head 上。

4. 这个小伙儿变帅了（虽然是光头）。我们借一副 Ethan 的眼镜并戴在头上。点击菜单 **GameObject | Create Empty**，重命名为 Glasses，用 **Transform** 组件的齿轮图标 | **Reset** 重置其 **Transform**，设置其 **Position** 为（0, –5.6, 0.1），**Scale** 为（4, 4, 4）。

5. 然后，当 Glasses 被选中时，在 **Project** 面板中，找到 Assets / Standard Assets / Characters / ThirdPersonCharacter / Models / Ethan / EthanGlasses.fbx（网格文件），把它拖进 **Inspector** 面板。确认选择 EthanGlasses 的 fbx 版本，而不是预制件。

6. 它有一个网格，但它需要一种材质。当 Glasses 被选中时，在 **Project** 面板中，找到 Assets / Standard Assets / Characters / ThirdPersonCharacter / Materials / EthanWhite，把它拖进 **Inspector** 面板。

下面的截图展示了我的版本（还有一个嘴巴）：



当多玩家运行时，将会为每位连接的玩家创建虚拟角色的实例。所以，我们首先必须保存这个对象为预制件，并从场景中将其移除，步骤如下：

1. 在 **Hierarchy** 面板中选择 Avatar，拖进 **Project Assets**。

2. 再次在 **Hierarchy** 面板中选择 Avatar 并删除。

3. 保存场景。

好了，现在我们应该准备好添加多玩家网络系统了。

10.3 添加多玩家网络

要让场景以多玩家方式运行，我们需要至少一个 Network Manager 组件，然后我们需要识别任何用 Network Identity 组件创建的对象。

10.3.1 Network Manager 和 HUD

首先，我们要添加 Network Manager 组件，步骤如下：

1. 点击 **GameObject | Create Empty** 并重命名为 NetworkController。

2. 点击菜单 **Add Component | Network | Network Manager**。

3. 点击菜单 **Add Component | Network | Network Manager HUD**。

我们还要添加一个 **Network Controller HUD** 菜单，一个 Unity 提供粗糙的默认菜单，用于选择运行时的网络选项（你可以在后面的图片中看到它）。它用于开发。在实际项目中，你应该用更合适的组件替换这个默认的 HUD。

10.3.2 Network Identity 和 Transform

接下来，添加一个 Network Identity 到 Avatar 预制件中。我们还将添加一个 Network Transform 组件，用于命令网络系统同步玩家的 Transform 值到每个客户端的虚拟角色实例，步骤如下：

1. 在 **Project Assets** 中，选择 Avatar 预制件。

2. 点击 **Add Component | Network | Network Identity**。

3. 勾选 **Local Player Authority** 复选框。

4. 点击 **Add Component | Network | Network Transform**。

5. 确认 **Transform Sync Mode** 被设置为 **Sync Transform**，而 **Rotation Axis** 被设置为 **XYZ（full 3D）**。

6. 现在，告诉 Network Manager 我们的 Avatar 预制件代表玩家。

7. 在 **Hierarchy** 面板中，选择 **NetworkController**。

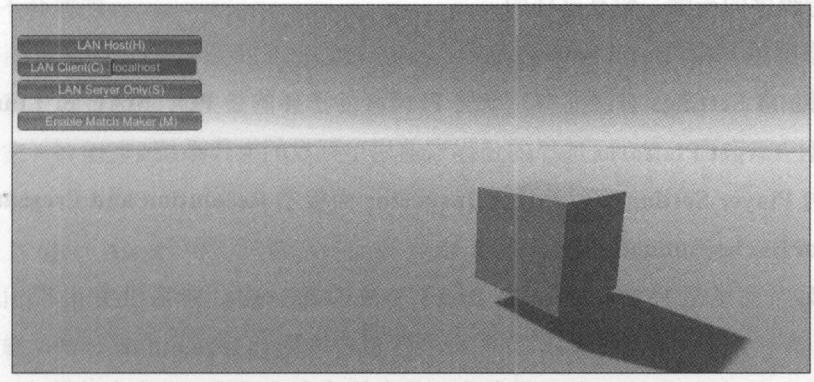
8. 在 **Inspector** 面板中，展开 **Network Manager Spawn Info** 参数然后可以看到 **Player Prefab** 槽。

9. 在 **Project Assets** 中，找到 Avatar 预制件并把它拖进 **Player Prefab** 槽。

10. 保存场景。

10.3.3　作为一个主机运行

点击 Play 模式。如下图所示，屏幕中出现 HUD 的开始菜单，可以让你选择是否愿意运行并连接到这个游戏。



选择 **LAN Host**，它将初始化一个服务器（localhost 上的默认端口 7777）并创建一个 Avatar。这个虚拟角色放在默认的位置（0, 0, 0）。另外，它没有连接到摄像机。所以，它更像是一个第三人称视角。

接下来要做的事情是运行第二个游戏的实例，然后在场景中看到两个创建出来的虚拟角色。然而，我们不想让它们重叠在初始位置上，所以首先需要定义一组出生点位。

10.3.4　添加出生点位

要添加出生点位，你只需要一个带有 Network Start Position 组件的游戏对象：

1. 点击菜单 **GameObject | Create Empty**，重命名为 Spawn 1，然后设置其 **Position** 为（0, 1.4, 1）。

2. 点击菜单 **Add Component | Network | Network Start Position**。

3. 点击菜单 **GameObject | Create Empty**，重命名为 Spawn 2，然后设置其 **Position**

为（0, 1.4, –1）。

4. 点击菜单 **Add Component | Network | Network Start Position**。

5. 在 **Hierarchy** 中，选择 NetworkController。在 **Inspector > Network Manager > Spawn Info**，**Player Spawn Method** 下，选择 **Round Robin**。

我们现在有两个不同的出生点了。Network Manager 将选择任意一个作为新玩家加入游戏时的出生点。

10.3.5 运行两个游戏实例

以一个合理的方式在同一台机器上运行两个游戏的复本是将一个实例作为一个单独的可执行程序构建和运行，而另一个实例以来自 Unity 编辑器的方式（Play 模式）运行。构建可执行程序的步骤一般是这样的：

1. 点击菜单 **File | Build Settings...**。

2. 在 **Build Settings** 对话框中，确保 **Platform** 选择的是 **PC，MAC & Linux Standalone**，并且 **Target Platform** 选择的是你当前用于开发的操作系统。

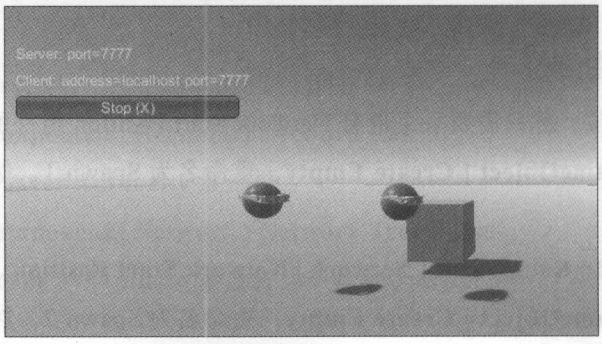
3. 点击 **Player Settings...**，然后在 **Inspector** 面板中 **Resolution and Presentation** 下，勾选 **Run in Background** 复选框。

4. 确保当前场景只在 **Scenes In Build** 中被勾选一次。如果没有出现，点击 **Add Current**。

5. 选择 **Build and Run**，起一个名称，并在构建完成后通过双击启动游戏。

启用 **Run in Background** 将允许用户在程序运行时的每个窗口输入（键盘和鼠标）。

在 Unity 编辑器中，点击 Play 模式并选择 **LAN Host**，就像之前那样。双击应用程序并勾选 **Windowed** 复选框让它不要全屏显示。然后，在程序窗口选择 **LAN Client**，在每个游戏中你都应该能看见两个虚拟角色实例，每个玩家有一个虚拟角色，如下图所示：



如果你想在一台单独的机器上运行游戏实例，在 **Client** 中输入主机的 IP 地址以替换 localhost（比如，我的 LAN 是 10.0.1.14）。

有一些键盘快捷键可以派得上用场。在 HUD 上，用 **H** 选择 Host，**C** 选择 Client，*Alt*+*Tab*（Mac 上是 Command+Tab）可以不用鼠标来切换窗口。

10.3.6　关联虚拟角色与第一人称角色

如果虚拟角色不移动的话就没什么意思，这是最后一道难题。

你可能以为我们应该让虚拟角色作为第一人称控制器 FPSController 的子对象，重命名并保存为预制件，然后选择 Network Manager 使用它创建对象。但是之后，你应该不再在场景中使用多个 FPSController 了，因为每个活跃的摄像机和控制器脚本都监听用户的输入是不合适的。

我们必须只有一个活跃的 FPSController、一个摄像机、一个输入控制器实例，等等。其他玩家的虚拟角色被创建但不是在这里予以控制。换句话说，当本地玩家（也只有本地玩家）被创建时，这个虚拟角色可以成为摄像机的子对象。要实现它，我们需要写一个脚本：

1. 在 **Project Assets** 中，选择 Avatar，点击 **Add Component | New Script**，命名为 AvatarMultiplayer.cs，并在 MonoDevelop 中打开。

2. 编辑 AvatarMultiplayer.cs 脚本，如下：

```
using UnityEngine;
using UnityEngine.Networking;

public class AvatarMultiplayer : NetworkBehaviour {

  public override void OnStartLocalPlayer () {
    GameObject camera = GameObject.FindWithTag ("MainCamera");
    transform.parent = camera.transform;
    transform.localPosition = Vector3.zero;
  }
}
```

第一件要注意的事情是我们需要包含 `UnityEngine.Networking` 库从而访问网络 API。然后，`AvatarMultiplayer` 类是 `NetworkBehaviour` 类型的，其本质上继承于 MonoBehavior。

`NetworkBehaviour` 提供了很多新的回调函数。我们使用 `OnStartLocalPlayer`，

它在创建本地玩家对象时被调用，但是它不会在创建远程玩家对象时被调用。它的声明需要 override 关键字。

OnStartLocalPlayer 正是我们所需要的，因为只有当本地玩家被创建时才需要把它作为摄像机的子对象。我们得到当前的 MainCamera 对象（GameObject.FindWithTag("MainCamera")）并让它成为虚拟角色的父对象（transform.parent = camera.transform）。我们还要重置虚拟角色的变换值，这样它的中心点就在摄像机的位置上了。

运行两个游戏实例——**Build** 并 **Run** 执行一个，Play 模式执行另一个。现在当你在一个窗口控制玩家时，它在另一个窗口中也移动。哇！你甚至可以启动更多的程序然后举办一个聚会了！

10.4 添加多玩家到虚拟现实

目前为止，我们已经学习了如何使用 Unity 实现多玩家网络系统，但不是特定于 VR。现在，我们为 VR 做准备。我们开始配置运行于 VR 的方法，不需要改变任何场景中的东西。

我们的虚拟角色因为某种原因只有一个头部。在 VR 中，摄像机的变换值由**头戴显示器**（HMD）传感器的头部动作控制。当虚拟角色作为摄像机的子对象时，它会同步移动。当你戴上 HMD，移动头部并向四周看时，所有客户端中相应的虚拟角色也会跟着移动。

10.4.1 Oculus Rift 玩家

目前场景中有一个标准的 FPSController 的实例，其中包括一个摄像机组件，我们可以使用 Unity 5 的内置 VR 支持。

在 **Build Settings** 对话框中点击菜单 **File | Build Settings...**，选择 **Player Settings...**。然后，在 **Other Settings** 下勾选 **Virtual Reality Supported** 复选框。

内置的 Network Manager HUD 是在屏幕空间中实现的。所以，如果你现在运行场景它是不会出现的。一个解决方案是在世界空间中实现你自己的 HUD 使游戏连接到网络。另一个解决方案是开启非 VR 模式并接连网络时在头盔上进行切换。我们将采取后一种方案，这需要两个小的脚本。

1. 在 **Hierarchy** 中选中 NetworkController，点击菜单 **Add Component | New Script**，命名为 NetworkStart，并在 MonoDevelop 中打开。

2. 编辑 NetworkStart.cs 脚本，如下：

```
using UnityEngine;
using UnityEngine.VR;

public class NetworkStart : MonoBehaviour {

  void Awake() {
    VRSettings.enabled = false;
  }
}
```

注意我们使用了 UnityEnigne.VR 类库。当游戏第一次初始化时，Awake() 被调用，我们可以在这里禁用 VR。现在，它将启动非 VR 模式。用户可以看到屏幕空间 UI 并选择自己的运行时网络选项。之后，我们在创建玩家时启用 VR。

接下来，打开 AvatarMultiplayer.cs 脚本并编辑它，如下：

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.VR;

public class AvatarMultiplayer : NetworkBehaviour {

  public override void OnStartLocalPlayer () {

    VRSettings.enabled = true;

    GameObject camera = GameObject.FindWithTag ("MainCamera");
    transform.parent = camera.transform;
    transform.localPosition = Vector3.zero;
  }
}
```

保存场景。

运行游戏的两个实例——**Build** 并 **Run** 启动一个，Play 模式启动另一个。哇哦，是一个 VR 聚会了！当你的头盔移动时，虚拟角色在每个客户端中移动。

在测试过程中，如果你像我一样只有一个 Oculus Rift（！），你可能只运行一个 VR 的实例。自然而然，提供一个带有选项的 **Start** 菜单来选择设备是一种方式。出于本例

的目的，让我们假设你只能把VR运行在Host实例上，其他客户端都不是VR。修改AvatarMultiplayer.cs脚本，如下：

```
public override void OnStartLocalPlayer () {
  if (isServer) {
    VRSettings.enabled = true;
  }
...
```

我们使用isServer检查当前的实例是否作为服务器运行（Host既是客户端也是服务器）。使用这样的技巧，Host实例可以使用Rift，非Host客户端则不能。

10.4.2 Google Cardboard 玩家

对于Cardboard，我们可以使用Cardboard自己的主摄像机预制件CardboardMain替代FPSController。我们将尝试以不影响任何现有运行的方式添加它，以便让同一个项目可以构建并运行在多个平台上而不用进行额外的修改。

添加Cardboard的主摄像机并禁用已经在场景中的FPSController，步骤如下：

1. 如果你没有加载Google Cardboard SDK包，点击菜单**Assets | Import Package | Custom Package...**，找到你下载的CardboardSDKForUnity.package，点击**Open**并选择**Import**。

2. 在Project Assets / Cardboard / Prefabs / CardboardMain中找到CardboardMain并拖进场景中。

3. 从FPSController中复制**Transform**的值或用**Transform**组件的齿轮图标 | **Reset**重置其**Transform**，再把**Position**设置成（5, 1.4, 3），**Rotation**设置成（0, 225, 0）。

4. 在**Hierarchy**中选择FPSController并通过反选**Enable**复选框禁用它。

打开我们之前创建的脚本文件NetworkStart.cs，在MonoDevelop中编辑，如下：

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.VR;

public class NetworkStart : MonoBehaviour {
  public GameObject oculusMain;
  public GameObject cardboardMain;
  public string hostIP = "10.0.1.14";
```

```
    void Awake() {
        VRSettings.enabled = false;

    #if (UNITY_ANDROID || UNITY_IPHONE)
        oculusMain.SetActive (false);
        cardboardMain.SetActive (true);
        NetworkManager net = GetComponent<NetworkManager> ();
        net.networkAddress = hostIP;
        net.StartClient ();
    #else
        oculusMain.SetActive (true);
        cardboardMain.SetActive (false);
    #endif
      }
    }
```

这段脚本为 oculusMain 和 cardboardMain 声明了 Public 的 GameObject 变量，我们将在编辑器中设置它们。

我不知道你使用的感觉如何，但是 Network Manager HUD 菜单在我的手机上操作起来太小了，尤其是在输入自定义 IP 地址时。所以，作为替代，我已经添加了一个内置的 IP 地址到脚本中。我的开发机作为主机运行在 10.0.1.14。

为了例子的简化和开发，在实际项目中，你需要在游戏中提供 UI 或使用 Unity 云服务。

你现在可以为 Android 系统构建并在你的手机上（安装 App）。（Cardboard 在 iOS 系统上的用法类似。）如同第 3 章中解释的那样，确认你的系统已配置好 Android 的开发环境。然后，切换项目的构建平台到 Android 系统中，如下：

1. 点击菜单 **File | Build Settings...**，在 **Build Settings** 对话框的 **Platform** 区域，选择 **Android** 并选择 **Switch Platform**。

2. 选择 **Player Settings...**，在 **Inspector** 面板中，确认你已经在 **Other Settings** 中设置了 **Default Orientation: Landscape Left** 和 **Bundle Identifier**。

3. 保存场景和项目，为 Android 构建一个 .apk 文件。

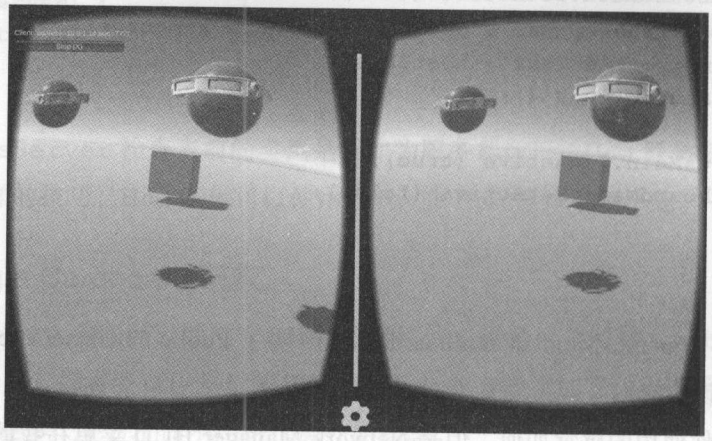
4. 切换平台并为 Windows 或 Mac 构建可执行程序。

5. 在每个平台上启动应用！聚会开始！

下图是我的 Android 手机中有关场景的截图。我让三个玩家的游戏运行——一个

Windows 以 Host 构建运行且使用一台 Oculus Rift，一台 Mac 作为客户端构建运行，一台 Android 以带有 Cardboard 的客户端构建运行。

同一个项目的三个单独的构建包分别在不同的平台上用不同的 VR 设备运行多玩家游戏。



10.4.3 下一步

前面是关于使用 VR 的 Unity 网络系统的介绍，而就这个话题则需要整整一本书。还有一些需要进一步阐释的领域如下：

- 构建一个自定义的世界空间 HUD 开始菜单，用于网络连接和设备选项。
- 通过 Unity Multiplayer 云服务连接各个客户端。
- 让用户自定义他们的虚拟角色，比如允许他们选择自己的肤色。
- 在他们的头上用一个信息框显示他们的用户名。
- 提供一个拥有完整身体的带动画的虚拟角色。
- 优化数据速率用于平滑移动。
- 添加聊天或语音通信。比如，看一下 Mumble（http://www.mumble.com/）和 TeamSpeak（http://www.teamspeak.com/）。
- 创建非玩家对象，比如一个弹力球，然后试玩用头部弹击的排球游戏！（参见 http://docs.unity3d.com/Manual/UNetSpawning.html 和章节 7.3。）

10.5 构建和共享一个自定义的 VRChat 房间

如果你的目标比较简单——构建一个虚拟现实世界并作为一种社交体验分享给其他

人，你可以使用现有的某些提供基础配套并允许自定义的社交 VR 应用。撰写本书时，这些应用包括 VRChat、JanusVR、AltSpaceVR、ConVRge、Vroom 和其他。

我最喜欢中的一个是 VRChat，我们将在下一个项目中使用它。VRChat 用 Unity 构建，并且你可以使用 Unity 制作自定义的世界和虚拟角色。如果你还没有试用过，可以从 http://www.vrchat.net/ 下载它的客户端副本来玩一下。（撰写本书时，你可以使用 Oculus Rift 进入 VRChat，但不能在手机上。然而，它却能在非 VR 的桌面模式下运行。）

在 VRChat 中选择一个场景试玩，选择任何一个你喜欢的 Unity 场景。它可以是我们之前在本书中使用的透视图游乐场，或者是第 8 章中的 PhotoGallery，或者是其他的。在 Unity 中打开你想导入的场景。

10.5.1　预备并构建虚拟世界

开始前，请从 http://www.vrchat.net/download 下载 VRChat SDK，并在 http://www.vrchat.net/docs/sdk/guide 查看其文档上的最新说明：

1. 导入 VRChat SDK 包，点击菜单 **Assets | Import Package | Custom Package...**，找到你下载的 VRCSDK-*.package，点击 **Open** 并选择 **Import**。

2. VRChat 需要对象的层级可能与你现在的场景的组织方式不同。所以，我建议你先通过菜单 **File | Save Scene As...** 保存一个新的复本，再取一个新的名字，比如 VRChatRoom。

3. 创建一个根对象，然后把你的房间对象作为其子对象。点击菜单 **GameObject | Create Empty**，重命名为 Room。选择所有 **Hierarchy** 面板中属于它的对象，并拖动它们成为 Room 的子对象。不要包含任何摄像机或其他你不想导入的东西。

4. 添加 VRC_SceneDescriptor 脚本到 Room。在 **Hierarchy** 面板中，选择 Room，然后点击菜单 **Add Component | Scripts | VRCSDK2 | VRC_Scene Descriptor**。

5. 创建一个位置点。点击菜单 **GameObject | Create Empty**，重命名为 Spwan，并移动它成为 Room 的子对象。在 **Scene** 场景视图中，拖进位置点。

6. 在 **Hierarchy** 面板中选中 Room，在 **Inspector** 中展开 Spawns 列表，把 **Size** 设置成 1，然后把 Spwan 对象拖进 **Element 0** 槽。

7. 导出此世界空间。在 **Hierarchy** 面板中选择 Room，再点击 **VRChat | Build and Run Custom Scene from Selection**。

你可以在 VRChat 进行预览。

10.5.2 承载这个世界

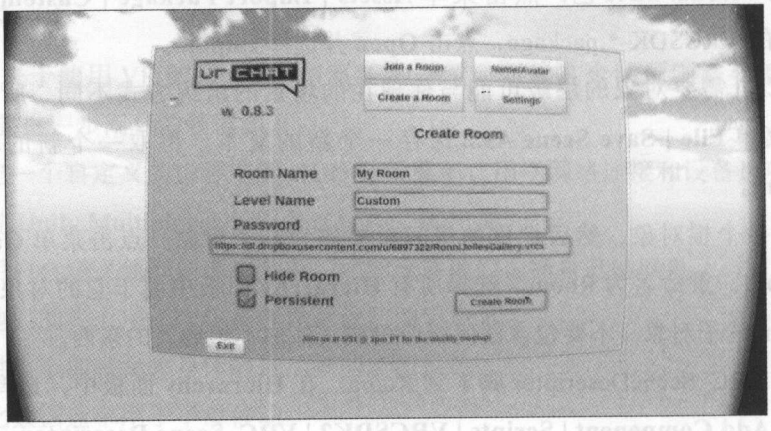
当你已经准备好在世界空间中分享你的房间时，你需要复制 built.vrcs 文件到网页可以访问的地方，比如一个公开的 Dropbox 文件夹或者一个网站服务器。你还需要粘贴文件的 URL，获取它的复本，执行下面的步骤：

1. 运行 VRChat 客户端。
2. 点击 **Create a Room** 按钮。
3. 点击 **Level Name** 下拉框并选择 **Custom**。
4. 粘贴并输入世界空间的 URL（比如 http://mywebsite.com/myWorld.vrcs）。
5. 如果你不想让你的房间出现在房间列表中，点击 **Hide Room**。

如果你不想在所有人离开你的房间后，你的房间消失在房间列表中，点击 **Persistent**。

6. 点击 **Create Room**。

下图展示了 VRChat 内部的对话框：



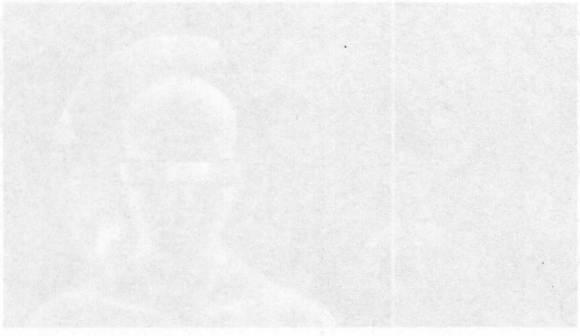
小结

本章中，我们学习了关于网络系统的概念和架构，然后使用了 Unity 自带的多玩家网络系统的某一些功能。我们构建了一个简单的场景和一个虚拟角色，目的是让虚拟角色的头部移动可以与玩家的头盔显示器同步。

之后，我们转换为多玩家场景，添加 Unity Network 组件，此组件将多玩家的实现简化成几次点击。

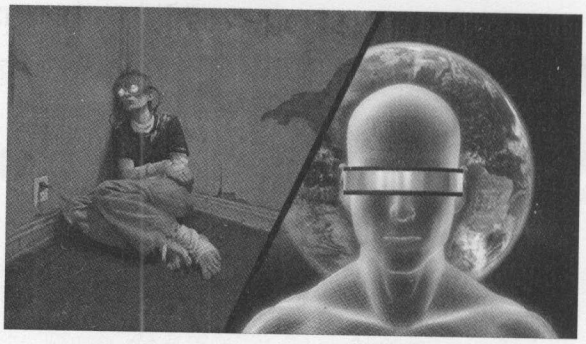
接着，我们把虚拟现实添加到了多玩家体验中，首先使用 Unity 对 Oculus Rift 的内置支持，然后通过添加 Google Cardboard 支持 Android。

最后，我通过导出一个几乎可以立即分享的场景向你们展示了用 VRChat 虚拟空间创建虚拟房间是多么的简单。



Chapter 11 第 11 章

虚拟现实的未来



Composite，Getty Images（书的封面）以及 VR 上瘾者的 Public Domain（源自 http://www.wallpapershome.com/art/graphics/wallpaper-cyberpunk-virtualreality-glass-addict-room-395.html）

不管你是从头到尾阅读此书，还是跳跃着挑选你想尝试的章节阅读，我都希望你或多或少学习了虚拟现实、Unity，以及如何构建自己的 VR 体验。这对于我们所有人来说都只是个开端，我们是先行者，你有机会去帮助创造未来。

我还有最后一点儿想法，希望比你所期待的技术书更富有哲理。

人类特别擅长在 3D 空间中行走并记住去过哪里。亲自去过的地方相比你只是听说，更有可能使你留住这段回忆。有已经被证实的记忆方法，比如记忆宫殿（memory palace），你通过在意识中构建一个虚构的用走廊和房间代表“存储”事情地点的城堡来记住一个晚宴或莎士比亚戏剧中的名字。此方法之所以能行之有效是因为它利用我们的

先天能力来记住空间。

使用这种本能的空间记忆，我们形成能够帮助我们解决问题的认识地图，这可能看起来太抽象或复杂了。这种功能已经不仅属于物理范畴，而且是工程学、数学、音乐、文学、视觉思维、科学、社会模型以及实际上大多数人类致力于的领域。我们着手于心理模型以获取对即将到来的问题的关键理解。

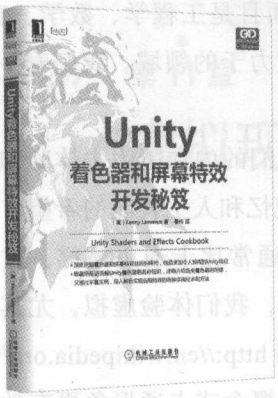
虚拟现实引人注目背后的一个原因是，因为它开拓了我们的内在能力去体验和使用三维空间而不受限于物理世界。VR 的应用给予了增强人类记忆和人工智能的希望。

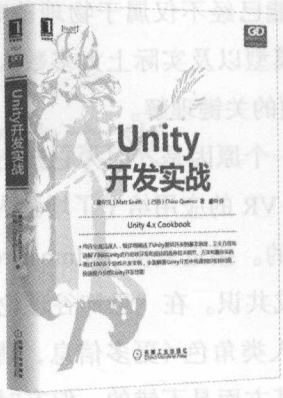
超现实主义是没什么必要的。接近真实（almost-real）通常比卡通更糟糕，这在被称为“恐怖谷理论”中已经达成共识。在“恐怖谷理论”中，我们体验虚拟，尤其是像令人毛骨悚然的并且非正常的人类角色（更多信息，请参阅 http://en.wikipedia.org/wiki/Uncanny_valley）。现实主义在某方面是不错的，但有时抽象概念或卡通形象更有效。请在设计我们自己的 VR 体验时记住这一点。

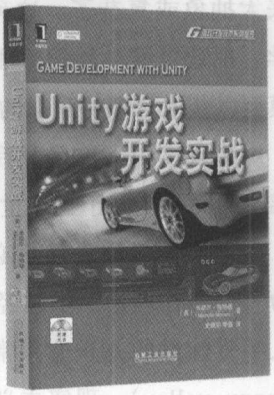
然而，很多人会期待在网络空间中有逼真的真实生活体验，至少是最终期望被虚拟现实所吸引。我认为这是一个错误，别以为这样的 VR 可以成功，我们必须体验一种对栩栩如生的虚拟环境进行完整的感官模拟。你想要带着食管沉浸在虚拟空间之中，就像《黑客帝国》或《玩家一号》那样？

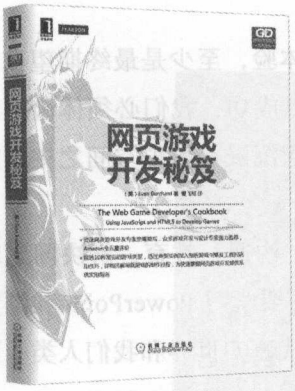
更有可能的是，当 VR 成为主流，它将发展成为表达、沟通、教育、解决问题和讲故事的媒介。你的祖父母需要学习打字和阅读，你的父母需要学习 PowerPoint 和浏览网页，你的孩子将在虚拟空间中构建城堡并传送。VR 不会替代真实世界和我们人类，而是会加强。

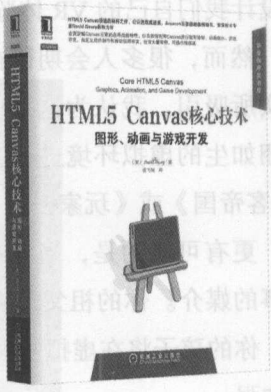
推荐阅读











Unity着色器和屏幕特效开发秘笈

作者：Kenny Lammers ISBN：978-7-111-48056-3 定价：49.00元

Unity开发实战

作者：Matt Smith 等 ISBN：978-7-111-46929-2 定价：59.00元

Unity游戏开发实战

作者：Michelle Menard ISBN：978-7-111-37719-1 定价：69.00元

网页游戏开发秘笈

作者：Evan Burchard ISBN：978-7-111-45992-7 定价：69.00元

游戏开发工程师修炼之道（原书第3版）

作者：Jeannie Novak ISBN：978-7-111-45508-0 定价：99.00元

HTML5 Canvas核心技术：图形、动画与游戏开发

作者：David Geary ISBN：978-7-111-41634-0 定价：99.00元

作者简介

Jonathan Linowes是处于起步阶段的VR/AR咨询公司Parkerhill Reality Labs的负责人，他是一位名副其实的VR和3D图形狂热分子、Web全栈工程师、软件工程师、成功的企业家，以及教师。他拥有美国雪城大学的美术学士学位和麻省理工学院多媒体实验室的硕士学位。他成功创办了多家创业公司并曾在大型公司从事技术工作，其中包括欧特克公司。

译者简介

童明，Polycom高级软件工程师，前微软最有价值专家（Windows平台开发方向），曾就职于联想集团北京公司，拥有多年的软件及互联网从业经验，具有丰富的客户端软件开发经验，译有《Unity开发实战》，著有《Windows 8应用开发实战》。

吴迪，Windows Phone高级软件工程师，拥有多年软件及互联网从业经验，现就职于Camera360，主导并参与Camera360 WP版及Windows 10版本。

封面设计：杨宇梅

什么是消费级虚拟现实？戴上一个头盔显示器（如护目镜），你可以观看立体3D场景，通过移动头部向四周看以及使用手持控制器或动作传感器向四处走动，你可以拥有一个完整的沉浸式体验。另外，Unity是一款强大的游戏开发引擎，提供了丰富的功能，比如视觉光照、材质、物理、声音、特效，以及创建2D和3D游戏的动画。Unity 5 已经成为领先的平台，为新一代消费级VR设备构建虚拟现实游戏、应用和体验。

通过阅读本书，你将学到：

- 使用Unity和Blender创建3D场景，学习世界空间坐标系和比例尺。
- 构建和运行针对消费级头盔设备的VR应用，其中包括Oculus Rift或Google Cardboard。
- 通过一个基于项目的指南学习使用Unity开发VR应用。
- 使用Unity引擎中的物理、重力、动画和光照构建交互环境。
- 实验各种你会在VR应用中用到的用户界面（UI）技术。
- 仅使用头部动作姿势作为输入实现第一人称和第三人称体验。
- 创建动画漫游，使用360°多媒体，并构建多用户的社交化VR体验。
- 学习有关VR的技巧和心理学，包括渲染、性能和VR晕动症。
- 在Unity中使用C#语言编程获取引导和进阶体验。

上架指导：计算机\游戏开发
ISBN 978-7-111-55131-7
9 787111 551317 >
定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn